Affect-Driven Self-Adaptation:

A Manufacturing Vision with a Software Product Line Paradigm

by

Javier Gonzalez-Sanchez

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved July 2016 by the
Graduate Supervisory Committee:

ARIZONA STATE UNIVERSITY

August 2016

ABSTRACT

Affect signals what humans care about and is involved in rational decision-making and action selection. Many technologies may be improved by the capability to recognize human affect and to respond adaptively by appropriately modifying their operation. This capability, named *affect-driven self-adaptation,* benefits systems as diverse as learning environments, healthcare applications, and video games, and indeed has the potential to improve systems that interact intimately with users across all sectors of society. The main challenge is that existing approaches to advancing affect-driven self-adaptive systems typically limit their applicability by supporting the creation of one-of-a-kind systems with hard-wired affect recognition and self-adaptation capabilities, which are brittle, costly to change, and difficult to reuse. A solution to this limitation is to leverage the development of affect-driven self-adaptive systems with a manufacturing vision.

This dissertation demonstrates how using a software product line paradigm can jumpstart the development of affect-driven self-adaptive systems with that manufacturing vision. Applying a software product line approach to the affect-driven self-adaptive domain provides a comprehensive, flexible and reusable infrastructure of components with mechanisms to monitor a user's affect and his/her contextual interaction with a system, to detect opportunities for improvements, to select a course of action, and to effect changes. It also provides a domain-specific architecture and well-documented process guidelines, which facilitate an understanding of the organization of affect-driven self-adaptive systems and their implementation by systematically customizing the infrastructure to effectively address the particular requirements of specific systems.

The software product line approach is evaluated by applying it in the development of learning environments and video games that demonstrate the significant potential of the solution, across diverse development scenarios and applications.

The key contributions of this work include extending self-adaptive system modeling, implementing a reusable infrastructure, and leveraging the use of patterns to exploit the commonalities between systems in the affect-driven self-adaptation domain.

To my dearest family

ACKNOWLEDGMENTS

I am deeply thankful to my advisor, Dr. Winslow Burleson for his strong support and guidance, for his kind encouragement in accomplishing this work, and for his career advice. My thanks to my committee members: Dr. David Garlan (Carnegie Mellon University) for accepting the role of external reviewer, for carefully reviewing my work and strengthening its quality, and for his feedback and willingness to help; Dr. Robert Atkinson for all of the lessons learned while collaborating on multiple research projects and for providing access to the many devices required for my research; Dr. James Collofello for his support in my research and teaching; and Dr. Hessam Sarjoughian for his help in focusing this work. My thanks to all for providing direction and clarity for my work.

My gratitude to: Dr. Kasia Muldner (Carleton University) for her friendship and lessons on research and writing; Dr. Paris Avgeriou (University of Groningen) for his valuable input in the early stages of this research; Dr. Kurt VanLenh for the lessons learned while collaborating on research projects; and Dr. Alfredo Hidalgo San Martin, Dr. Bettylu Rassmusen Cruz, Dr. Bertha Marquez Azua (Universidad de Guadalajara), Dr. Raul Jacinto Montes (CINVESTAV), and Dr. Juan Francisco Corona Burgueño (Tecnologico de Monterrey) for their inspiration and mentorship.

My work could not have been achieved without the members of the Motivational Environments research group, the Advancing Next Generation Learning Environments (ANGLE) Lab and the Affective Meta Tutor (AMT) group. They have been a great team of friends and colleagues to work with. They have my esteem and my gratitude. My appreciation also to the many contributors to the experiments, particularly the ASU students who chose to do their capstone project under my supervision and the ANGLE Lab staff for assisting me with the capstone student mentoring and management.

I would like to acknowledge the help of the attendees and organizers of the doctoral consortium at the 2013 ACM/IEEE International Conference on Software Engineering (ICSE) for their advice and the exchange of ideas, which helped me to mature this work. I would also like to acknowledge the ASU-CONACYT Fellowship Program for funding the first four years of my doctoral research.

I am eternally grateful to my dear family; especially to my mother, Teresa Sanchez, for her life-long support and love. And, I deeply appreciate the moral support of my friends, all of whom I cannot name here. My heartfelt thanks to you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

CHAPTER 1
INTRODUCTION

Affect (emotions, feelings, and moods) is inextricably bound to human cognitive processes and expresses a great deal about human necessities (Picard 1997). Today, we know that affect signals what matters and what humans care about and, furthermore, that affect is involved in rational decision-making and action selection (Picard 2010). However, most systems today lack affective capabilities and provide cumbersome and rigid interactions that do not emulate the naturalness, flexibility, and robustness of human-human communication (Maat & Pantic 2007; Cheng et al. 2009; Calvo & D'Mello 2010; Van den Broek 2011; Camara, Moreno, & Garlan 2015). An emerging trend aims to tackle these limitations by providing systems with *affect-driven self-adaptation* – the ability to recognize users' affective state changes, understand the meaning of these changes, learn from them, and use them to drive real-time reactions that improve system behavior. Affect-driven self-adaptation provides principled and automatic adaptation in a human-centered context, advancing system functionality and improving human-computer interactions in general and user-experience in particular. Learning environments, healthcare systems, and video games are just a few of the types of systems that stand to benefit from affect-driven self-adaptation to better support their purpose: coaching, companionship, entertainment, etc. For instance:

a) An intelligent tutor detecting students' affect can realize and respond to a student's need for emotional support. Evidence indicates learning is enhanced when empathetic intervention or support is present (Arroyo et al. 2009; Lehman, D'Mello, & Person 2008). The intelligent tutor can use affect, such as boredom, frustration, and excitement, to provide encouraging comments or to alter the level of support (feedback and hints), as well as task difficulty.

b) A video game can become more compelling by using players' affect as input to alter and adjust the gaming environment. Affect, such as excitement or fear, can be used by the game to alter lighting, music, colors, complexity, or level of companionship.

c) An avatar in a virtual world can mirror a human's affective expressions. Mirroring affect helps the avatar become more believable, likeable, trustable, and enjoyable, as well as creating long-lasting relationships (Bickmore & Picard 2005; Burleson & Picard 2007).

d) A healthcare application can provide empathetic interventions and motivational support to offer assistance and empower patients to improve their quality of life. Affect, such as frustration and confusion, can trigger interventions.

The motivation behind this doctoral research is to advance the creation of affect-driven self-adaptive systems by supporting an approach that empowers the broader deployment and increased adoption of the affect-driven self-adaptation capability.

1.1. Challenges

Imagine that software engineers could provide a new system with an affect-driven self-adaptation capability or could take an existing software system and improve it by augmenting it with this capability, where there was none before, without having to rewrite the system from scratch. Now, imagine the engineers could share and reuse tools, strategies, or implementation assets to achieve this task. Furthermore, imagine that these engineers could achieve their tasks within a few days or weeks, rather than weeks or months, without interfering with the existing system functionalities and properties. Achieving this level of deployment for the affect-driven self-adaptation capability in software systems involves the following three main challenges:

a) *Affect as a driver of self-adaptation*. Although the feasibility of developing affect recognition systems has been demonstrated (Picard 1997), most affect recognition efforts have been geared toward offline analysis of affect (creation of mathematical models). By and large, they have not overcome the challenge of advancing, at-scale, software systems working with users in real-time, without human supervision, in which affect recognition could trigger self-adaptation of the system.

b) *Explicit modeling of human factors in self-adaptation*. To date, self-adaptation models mainly deal with performance, resources, and error recovery as variables that drive system reactions. Self-adaptation based on human factors, such as affect measurement, present a different kind of challenge. Cheng et al. (2009) state that incorporating the user as an entity in self-adaptation models (analyzing feedback from human-computer interaction) is an issue that needs to be addressed.

c) *Craftsmanship versus manufacturing.* Existing approaches to creating affect-driven self-adaptive systems limit their applicability by supporting craftsmanship of one-of-a-kind systems whose purpose is to verify that a specific concept or theory is possible. Few models, frameworks, libraries, or software tools are available to allow system designers and developers to create or integrate the affect-driven self-adaptation capability (or even just affect recognition) into their software projects; therefore, most affect-driven self-adaptive systems deployed today have hard-wired affect recognition and self-adaptation capabilities. Moreover, the affect recognition and adaptation logics are often dispersed and entwined throughout the system, making it brittle (difficult to modify and maintain) and making reuse nearly impossible. Thus, developing new systems with the affect-driven self-adaptation capability requires significant duplication of effort and, consequently, high cost. Cheng, Garlan, & Schmerl (2005) document this problem for engineering self-adaptive systems in general and Clay et al. (2009) point out a similar problem for engineering affect recognition in systems. Efforts in the manufacturing of affect-driven self-adaptive software are rare, and adapting craftsmanship approaches to the demands of manufacturing has not yet been successfully demonstrated.

1.2. Opportunities for Improving the State-of-the-Art

Engineers and researchers have responded to and made some progress in addressing the need for affect-driven self-adaptation, albeit somewhat limited. A number of researchers have explored self-adaptation using affect to drive real-time reactions with varying success, and systems with the affect-driven self-adaptation capability have been implemented. For instance, in the Intelligent Tutoring System community, affect-driven adaptation support has gained ground in custom-made solutions. On several occasions this community has explored the use of affect awareness to improve their systems. These efforts have proven the benefit of creating an affective connection with users, by modifying system functionality driven by sensing and responding to changes in users' affective states.

However, in general, the elements required for building adaptation based on human factors (such as affect measurement) have not been systematized and most research neither has a manufacturing vision nor follows (or reports to follow) software engineering models or methodologies. As a result, affect-driven self-adaptation in today's systems is costly to build, often taking many man-months to develop or retrofit systems with these capabilities. Moreover, once added, the capabilities are difficult to modify and usually provide only limited treatment of affect recognition and adaptation.

Chapter 2 describes the state-of-the-art referencing related work; as a summary, the affect-driven self-adaptation capability is often:

- proof-of-concept oriented and focused on a fixed set of affect-sensing devices, inference mechanisms, and quality concerns;
- handcrafted with the affect recognition and adaptation logics dispersed and entwined throughout the system implementation, making their reuse nearly impossible;
- complex, requiring significant developer involvement in gaining affect recognition and self-adaptation expertise as they endeavor to implement this capability; and
- costly to develop, both for new systems and to evolve modifications and maintenance for existing systems.

While there have been meaningful contributions to the general problem space of affect-driven self-adaptation, there remain significant limitations. These limitations can be best understood in terms of three fundamental issues: generality, feasibility, and cost-effectiveness. Particular aspects of these limiting issues can be addressed through three key elements: a domain-specific model or architecture, a reusable infrastructure, and process guidelines. The use of a model and an infrastructure support generalization, the use a model and guidelines increase feasibility, and the use of an infrastructure and guidelines improve cost-effectiveness. The relationship between the three limiting issues, the aspects of these that are addressed, and the three elements used to do so is depicted in Figure 1.1 and described in the following subsections.

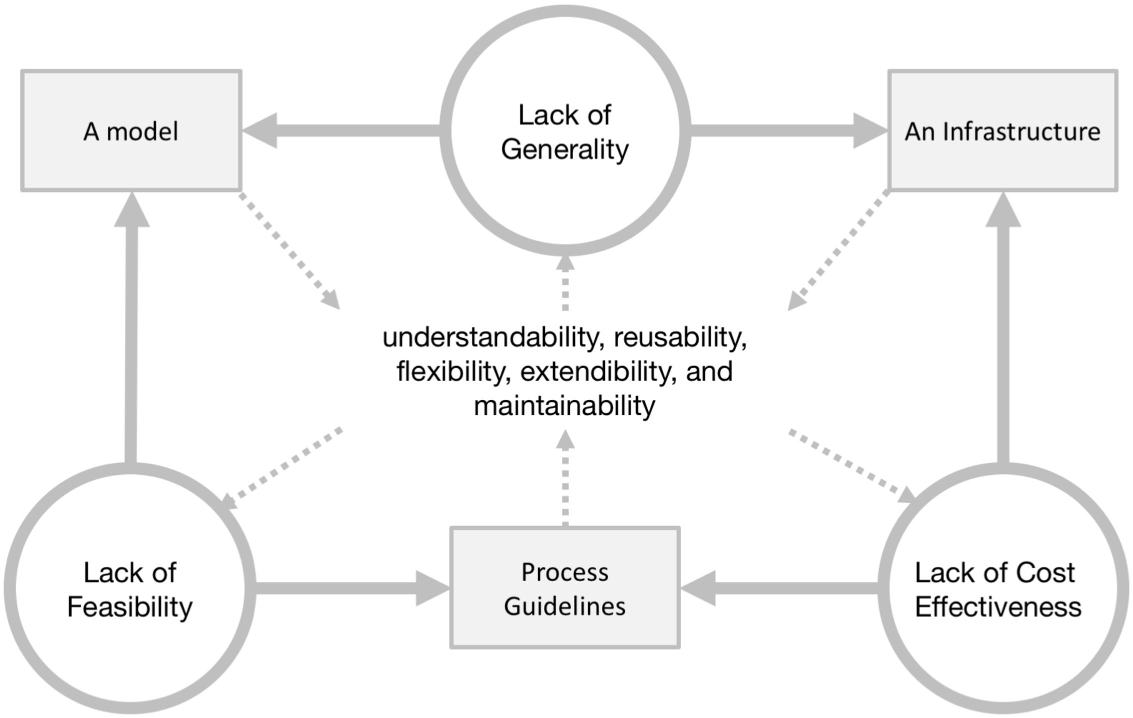

A model
Lack of Generality
An Infrastructure
understandability, reusability, flexibility, extendibility, and maintainability
Lack of Feasibility
Process Guidelines
Lack of Cost Effectiveness

Figure 1.1. Relationships between limiting issues, aspects addressed, and elements that impact their improvement.

1.2.1. Lack of Generality

Lack of *generality* refers to a limited degree of function or range of application. Currently we can dynamically recognize affect and share this information with target systems. Also, from machine learning and decision theory communities, we have techniques to analyze what changes to make, as well as to reason about what course of action to take. However, we are missing the following crucial elements:

a) An explicit structural model to integrate affect recognition and adaptation capabilities into the system context or execution environment. A model able to provide a systematic and explicit representation of affect recognition and adaptation logics organization. The lack of an explicit structural model creates blind spots in the system engineering preventing an understanding of the process and the parts involved. Furthermore, this lack of explicit abstractions makes it complicated to reason about affect-driven self-adaptation, therefore limiting the use of available resources and strategies for their implementation.

b) An implementation of reusable assets that encapsulates affect recognition and adaptation existing expertise. The lack of a systematic and explicit encapsulation of affect recognition and adaptation strategies into assets makes it difficult to capitalize on expertise already available today, such as real-time affect recognition techniques and computational models for handling adaptation rules.

Addressing these limitations requires:

a) A structural model that allows software engineers to reason over the affect-driven self-adaptation, the core and optional tasks required, the organization of the steps involved in each task, and the affect-driven self-adaptive system design choices and decisions.

b) Software components that encapsulate existing expertise in affect recognition and adaptation.

1.2.2. Lack of Feasibility

Lack of *feasibility* refers to the limited degree to which the requirements, design, or plans for a system can be implemented under existing constraints. Feasibility is related to the knowledge that the engineers have of the process to be implemented. The lack of information usually means that implementation decisions and trade-offs will be affected by the competency of the engineer(s), who might not be familiar with affect or adaptation topics. Robust and generalizable guidelines and documentation are needed, including:

a) Guidelines for designing a self-adaptive system, which help to (1) separate concerns for adaptation, thus making the distinct tasks and steps of the process understandable to the engineers; and (2) realize adaptation concepts, thus making actions individually definable and composable to achieve overall system goals.

b) Guidelines for assembling an affect recognition logic, including selecting and balancing multiple, possibly conflicting, objectives such as accuracy versus performance. For instance, in the context of a game or intelligent tutor, providing a quick response could be as important or even more important than considering affective support for that response. Having the system freeze or be inactive (waiting for an internal process to finish before responding to the user) for a long time could provide a worse user experience than the traditional user experience without affective support or awareness. A naive implementation of decisions and trade-offs means systems will suffer from a number of disadvantages.

c) Documentation about resources. Current approaches address only fixed sensing devices and emotional state models, have hardwired responses, lack support for multimodal composition and trade-offs, lack resource selection, and do not cater to varying systems' contexts. Thus, adding a new sensing device may be difficult, requiring changes that affect many aspects of the system, including those that integrate the affect measurements and those that compute trade-offs and decide an outcome.

Addressing these limitations requires a systematic approach, along with guidelines and documentation for how best to couple resources and techniques, in ways that:

a) allow independence of affect measurement and multimodal integration from the adaptation process;
b) facilitate reasoning about and composing multimodal affect recognition that trades-off across multiple dimensions to produce the desired objectives, enabling flexible modification of sensing devices and affect measurement integration algorithms;
c) enable flexible modification of adaptation rules or preferences to cater to evolving contexts.

<u>**1.2.3. Lack of Cost-effectiveness**</u>

Cost-effectiveness refers to high cost of both development and maintenance. Building an affect-driven self-adaptive system is a costly proposition because one has to develop the infrastructure for sensing user signals, infer affect measurements, map them to a common reference space, and fuse them, as well as create an infrastructure to monitor the target system, a representation to encode constraints, a problem detector to determine adaptation opportunities, and adaptation mechanisms to resolve problems and propagate changes to the system. All of these parts must also be integrated into a coherent system. Most existing approaches have associated toolsets targeting a single sensing device and fixed adaptation concerns, often sacrificing system quality. Applying these approaches to a new system, often with a different functionality or set of concerns, would require either re-developing the affect awareness or adaptation mechanisms from scratch or reusing some parts and custom-building the remaining capabilities. Both incur significant time and effort to develop, integrate, and engineer the adaptation mechanisms appropriately. Once built, modifying or enhancing the sensing and adaptation mechanisms, likewise usually entail significant effort. In short, whether developing a software system from the ground up with affect-driven adaptation support or retrofitting an existing system, both options are usually quite costly. Reducing the costs associated with developing affect-driven adaptation support requires the following:

a) An infrastructure that amortizes cost and effort across multiple systems, enabling engineers to avoid re-developing significant mechanisms and that allowing low-effort incremental development of affect recognition and affect-driven adaptation capabilities.
b) Extensive guidance and documentation that offer a path to a solution with low risk and high payoff; particularly since a special interest exists in supporting the adoption of the affect-driven self-adaptation capability in communities where developers' experience or development resources are limited (such as research communities in education technology).

1.3. Hypotheses

To improve the state of current practice and overcome the limitations outlined above, this dissertation investigates a solution that takes advantage of affective computing research, self-adaptive system models, and advances in software engineering methodologies. The thesis discussed in this work addresses the challenges and limitations above and is stated as follows:

It is possible to support a broad creation and integration of affect-driven self-adaptation as a core capability of both new and pre-existing systems in a general, feasible, and cost-

effective manner, while promising adequate software quality (particularly in relation to the attributes of understandability, reusability, flexibility, extendibility, and maintainability) by defining and developing an approach that:

a) *takes advantage of current affect recognition tools, such as the ones described in Calvo & D'Mello* (2010)*, and adapting affect recognition strategies to drive the adaptation process;*
b) *extends current self-adaptation models, such as the ones described in Cheng, Garlan, & Schmerl* (2005) *for performance-driven and resource-driven self-adaptation, to support affect-driven self-adaptation; and*
c) *adopts a manufacturing vision where affect-driven self-adaptive systems could be developed from a common and managed set of core assets following proven practices that guide systematic asset assembly and integration* (Greenfield & Short 2003).

1.4. Approach

To do so, I propose using a software product line (SPL). An SPL supports a manufacturing vision capturing and encoding knowledge and proven practices to allow systematic development of systems that share a common set of features (Clements & Northrop 2002). The rationale behind using SPLs takes into consideration the following characteristics of SPLs:

a) SPLs implement a *manufacturing vision* focused on satisfying the needs of a particular market segment or mission – in this case, the mission to provide affect-driven self-adaptation.

b) SPLs enable the capture and encoding of knowledge and proven practices, allowing developers to *take advantage of and extend current tools and models* – an SPL for affect-driven self-adaptive systems captures and encodes expertise associated with affect recognition and self-adaptation domains.

c) SPLs favor *generality*. They use a systematic reusable infrastructure applicable to diverse kinds of systems. Their intent is to apply to a broad spectrum of systems without interfering with defined functionalities while retaining quality and trade-offs.

d) SPLs favor *feasibility*. They use a domain-specific model and production guidelines to make the processes understandable, actions composable, and choices automatable.

e) SPLs favor *cost-effectiveness*. Compared with custom, style-specific or empirical solutions, an SPL significantly reduces the cost to engineer and evolve systems.

f) SPLs offer a *low-risk and high-payoff* practice, implementing systematic reuse aimed at improving productivity (cost and time) and quality.

Moreover, SPLs have been reported as useful in many different domains, including smart building systems (Possompes et al. 2010), space flight software (Fant, Gomaa, & Pettit 2012), runtime interoperability (Siegmund et al. 2009), autonomic pervasive systems (Cetina, Fons, & Pelechano 2008), and mobile middleware (Morais, Burity, & Elias 2009); likewise, SPLs have been applied to address some aspects of adaptation (Hallsteinsen et al. 2008; Shen, Peng, & Zhao 2012).

Following the SPL paradigm, a manufacturing vision can be defined for software with affect-driven adaptive capabilities. This affect-driven self-adaptive software product line (ADASPL) can: (1) support a broader creation and integration of affect-driven self-adaptation; (2) enforce aspects of generality, feasibility, and cost-effectiveness; and (3) allow developers to take advantage of current assets and to extend current models.

The SPL paradigm is further described in Chapter 2. However, at this point, it is worth establishing that the application of the paradigm to a domain area, such as affect-driven self-adaptation, leads to a three-part technical solution. This three-part solution

a) formalizes the structure and organization of affect-driven self-adaptive systems;
b) establishes an infrastructure for affect-driven self-adaptive systems by providing core, optional, and customizable components that encapsulate common functionalities;
c) defines and documents a process for manufacturing affect-driven self-adaptive systems.

1.5. Contributions

This dissertation advances the state-of-the-art in software engineering and human-computer interaction by improving the understanding of the affect-driven self-adaptation capability, its design issues, and the mechanisms for its deployment with a manufacturing vision, demonstrating a more expansive coverage of the affect-driven self-adaptation problem space. The key contributions of this dissertation include the following:

a) Extending related work in self-adaptive systems. It defines an architectural model that provides a useful set of abstractions to focus engineers on affect-driven adaptation concerns, facilitating the systematic customization of a reusable infrastructure to advance particular systems. I follow software architecture methodologies, such as those described by Bass, Clements, & Kazman (2003) and Fairbanks (2010), to define an architectural model that defines how systems must be built, advancing common problems and generalizing them.

b) Implementing a reusable infrastructure, formed by a collection of customizable components and interfaces, to create a framework for Affect-Driven Self-Adaptive Systems development, named ADAS. I developed ADAS with core and optional components that can be included, extended, and associated with the rest of the software in the system by following object-oriented practices. ADAS provides a broadly applicable and reusable infrastructure with well-defined customization points to cater to a wide range of systems. It provides affect recognition and adaptive functionalities to monitor user affective states and target system status, fuse and classify affect, detect opportunities for improvements, select a course of action, and effect changes.

c) Leveraging the use of patterns as templates to exploit commonality between systems and to document guidelines that specify objectives, properties of interest, and articulate strategies for implementation. The guidelines recommend a workflow to tailor ADAS for specific systems. They provide step-by-step solutions to common necessities and so allow engineers to focus on system functionalities, adaptation strategies, or affective computing algorithms improvement.

d) Demonstrating the efficacy of ADASPL in representative areas of the affect-driven self-adaptation domain (learning and video gaming) by developing a set of systems and conducting a series of evaluations.

Figure 1.2 visualizes the approach and its constituent parts: the SPL paradigm applied to the affect-driven self-adaptation domain creates ADASPL, which defines a technical solution composed of an architectural model, an infrastructure (ADAS framework), and process guidelines. ADASPL does not engage with the organizational specifics. Collectively this dissertation provides novel and useful insights that have strong potential to influence the way in which software engineers design a broad range of next-generation human-centered software systems.

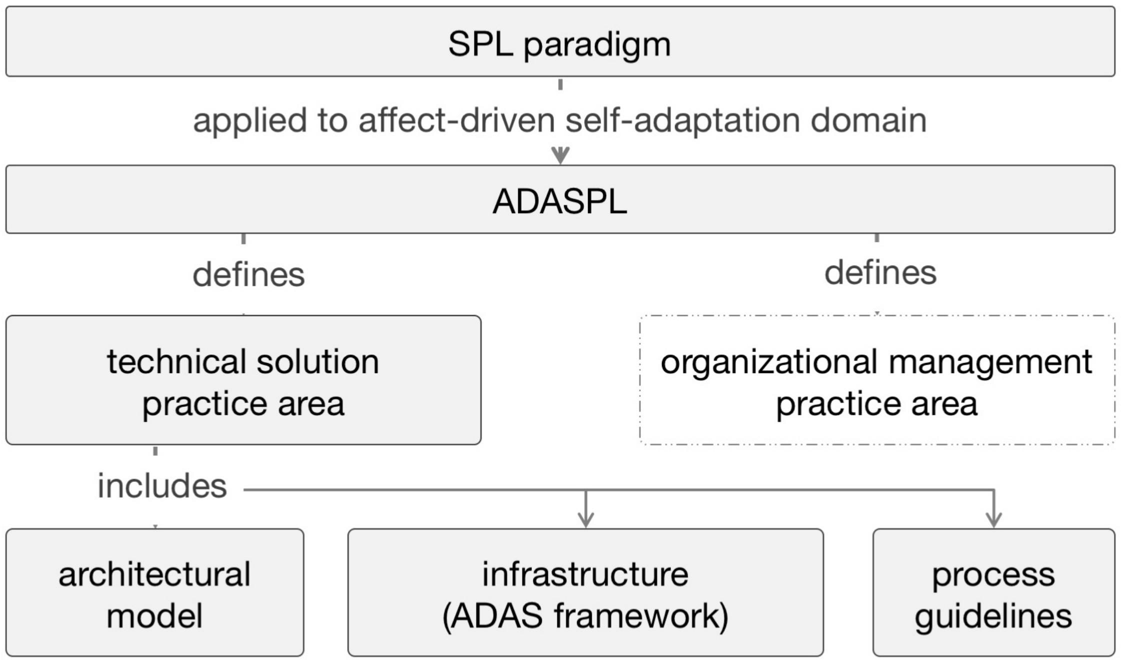

SPL paradigm
applied to affect-driven self-adaptation domain
ADASPL
defines
defines
technical solution practice area
organizational management practice area
includes
architectural model
infrastructure (ADAS framework)
process guidelines

Figure 1.2. Key constituents of the approach.

1.6. Evaluation Plan

An SPL evaluation involves realizing an in-depth demonstration of the SPL, applying the common and reusable infrastructures and customizing pieces for representative systems while running a validation of its parts and an empirical evaluation of the operation. For this research, each of these is accomplished as follows:

a) *In-depth demonstration.* To demonstrate the SPL, I focus on an important subgroup of computing systems requiring an enhanced human-centered approach, learning and video gaming systems. Eleven separate systems were variously developed by research teams and by undergraduate students involved in capstone projects.

b) *Validation of software assets.* The validation of the software assets includes the use of techniques for analyzing the architecture and the use of software testing methods to validate the framework components. On the one hand, an analysis of the architecture reveals trade-offs and sensitivity points and could be conducted following the Architecture Tradeoff Analysis Method (ATAM) (Bass, Clements, & Kazman 2003). However, architecture analysis is not included in this dissertation since the architectural model is an adaptation of related models applied in the self-adaptation field. On the other hand, an analysis of the framework confirms framework functionalities and qualitatively demonstrates framework generalization across different concerns and styles of systems (i.e., broad deployment). A framework can be evaluated using a dedicated test bed with case studies of systems with diverse concerns, as previously done by Cheng, Garlan, & Schmerl (2005) to validate their self-adaptive framework. Additionally, the products assembled using the provided assets can be evaluated using software metrics to measure their code complexity and structural qualities.

c) *Evaluation of the operation.* The empirical evaluation of the SPL operation is qualitative and quantitative. On the one hand, qualitative evaluation shows whether the SPL is understandable (engineers are able to use the architecture, customize the framework, and follow the guidelines) and makes it feasible to deploy the affective-driven self-adaptation capability. Understandability is evaluated by interviewing developers and analyzing how well they applied the SPL process. Data is collected from the developers' experiences to inspect the balance between the simplicity and the power of the framework, i.e., to validate whether the practice provides low risk and high payoff. On the other hand, to show these savings quantitatively, a cost-benefit analysis is performed. This is an important validation of the SPL as it demonstrates that developing new systems with the affect-driven self-adaptation capability in a traditional way requires a significant duplication of effort and, consequently, higher cost. For a cost-benefit analysis, the implementation tasks should be characterized to provide coarse-grained task-based estimation of effort, keeping track of development time and then qualitatively assessing savings relative to traditional practices of implementation.

1.7. Scope of Work

Several topics are beyond the scope of this dissertation, including:

a) *Security and privacy concerns.* Sensing affect raises critical privacy concerns and involves ethical and moral decisions and implications. Reynolds & Picard (2004) address the importance of considering the inclusion of ethical contracts to ground respect for privacy. While it was not incorporated in the materials used by the teams in this dissertation, it will be included in the broader dissemination of ADASPL.

b) *Affect recognition devices and techniques.* A description of some of these unconsidered topics is summarized in Calvo & D'Mello (2010) and includes sensing device development; selection of features; classification techniques; fusion methods; and performance, robustness, and accuracy of algorithms and tools.

c) *Include affect in the engineering process.* Elicit, analyze, specify, and validate affect requirements, affect testing process, affect maintenance, affect configuration management, and other affect-specific parts of the software engineering process. For instance, Callele et al. (2006) have researched the documentation of emotional requirements for video games.

d) *Include adaptation in the engineering process.* Elicit, analyze, specify, and validate adaptation requirements, adaptation testing process, adaptation maintenance, adaptation configuration management, and other adaptation-specific parts of the software engineering process. A number of existing approaches provide support for this crucial task, most of which are based on the use of temporal logic (Alrajeh et al. 2009).

e) *Organization management*. Although an approach based on the SPL paradigm is proposed and this paradigm covers both technical and organizational areas, this research takes advantage of technical solution strategies only and does not engage with the organizational specifics.

f) *Self-adaptation for learning systems.* Self-adaptive learning systems aim to fully create an instructionally sound and flexible environment adapting to abilities, disabilities, interests, backgrounds, and other characteristics (Shute & Zapata-Rivera 2007) beyond affective responses. For example, Chi et al. (2010) refer to monitoring user-events, such as measuring mistakes, the number of times the student gives up, or the number of times the student answers correctly, to create a user-model that drives self-adaptation.

Nevertheless, topics a, c, and d are considered further as part of the future work described in Chapter 10.

1.8. Document Roadmap

In this dissertation, I make a case for affect-driven self-adaptive systems, survey the research landscape, introduce an SPL approach, describe how the approach addresses current limitations and achieves the stated goals, discuss the research and engineering challenges, and demonstrate the approach with sample applications focusing on adaptations to improve learning and video gaming systems. Chapter 2 establishes the context by summarizing the terminology and the state-of-the-art of the three topics encompassed in this dissertation: self-adaptive system engineering, affective computing, and SPLs; it also surveys the research landscape and discusses related solutions and their limitations. Chapters 3, 4, 5 and 6 describe the overall research approach for the dissertation, and how the approach addresses current limitations and attempts to achieve the stated goals. Chapter 7 illustrates the approach to customization for a set of target systems, while Chapter 8 analyzes the process applied and systems created in order to quantify the evidence in support of this dissertation's arguments. Chapter 9 evaluates and discusses results, issues and limitations, and Chapter 10 provides conclusions, highlighting directions for future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter introduces self-adaptation, affect recognition and the SPL paradigm. It summarizes their background and their state-of-the-art. This chapter also presents the process measures and software metrics used to assess the SPL's use and the products created with it. It then moves on to discuss research work relevant to affect-driven self-adaptation, highlighting the foundational elements provided by the state-of-the-art and the state-of-the-art limitations addressed in this dissertation.

2.1. Self-Adaptation and the Closed-Loop Paradigm

Increasingly, modern systems must operate in the face of change and are also expected to accommodate themselves to those changes at run time with minimal human oversight. A system is said to be self-adaptive if it decides autonomously how to adapt or organize to accommodate changes in its environment (Cheng et al. 2009; Gowri et al. 2010) in order to sustain or improve the system's utility.

A self-adaptive system takes into account two elements: (1) system or environment changes that need to be monitored and (2) an adaptation strategy that defines the response of the system. Research on self-adaptation has dealt with diverse types of changes, classified into four categories (Norvig & Cohen 1997; Stavru 2011): variability in goals and trade-offs (e.g., performance), hardware and software resources (e.g., bandwidth and service availability), system faults (e.g., server components failing or connections going down), and individual differences among users (e.g., level of knowledge, skills, attention, preferences, moods, or intentions). Strategies deployed for self-adaptation include, but are not limited to, on-demand service consignment, load balance tactics, recovery procedure execution, and (where adaptation is related to users) maintaining for each user his/her own personal basis for driving the desired objectives, providing assistance, adding or removing the automation of tasks, changing information presentation, and increasing or decreasing typicality.

Since the late 1990s, multiple perspectives on self-adaptive software systems have been advanced and several development solutions have been proposed, among them model-driven self-adaptation (Cheng, Garlan, & Schmerl 2009). Model-driven self-adaptation has been reported to be one of the most promising options, particularly model-driven approaches based on control theory (Filieri et al. 2015) and in particular on the closed-loop paradigm. The closed-loop paradigm considers purely algorithmic models to break down when the execution of the system is affected by external disturbances – events that are not directly visible to or controllable by the software (Shaw 1995). The closed-loop paradigm assumes self-adaptation as a control problem with a looping three-step solution methodology:

a) Monitoring changes in the environment and the system and then determining if something has gone awry, usually because the change exhibits a value outside expected bounds or exhibits a degrading trend.
b) Determining a course of action to adapt the system once a problem is detected; thus, calculating corrective actions according to the objective of the adaptation and the strategy to follow.
c) Acting to carry out a chosen course of action and effecting the changes in the system.

Norvig & Cohen (1997) refer to these three steps as “perception,” “reasoning,” and “action”; Ramirez & Cheng (2010) name them “monitoring,” “decision-making,” and “reconfiguration”; Cheng, Garlan, & Schmerl (2005) name them “monitoring,” “modeling,” and “controlling”; and the IBM MAPE-K (Jacob, Lanyon-Hogg, Nadgir, & Yassin, 2004; Kephart & Chess, 2003) refers to them as “monitor,” “analyze and plan,” and “execute.” Norvig & Cohen (1997) point out that implementing this three-step solution methodology requires consideration of the following:

- Interfaces to gather information from the environment and the system.
- Interfaces to make adjustments to the system.
- Design and implementation of alternative functionality choices.
- Design and implementation of the decision-making rationale that enables the system to reconfigure itself at run time.

Shaw (1995) proposed an architectural model to represent closed-loop systems, which isolates the solution for the adaptation problem and the target system that requires adaptive capabilities. Shaw’s model defines two parts: an adaptation logic and a functional logic. The adaptation logic monitors changes that drive the adaptation process, and calculates and triggers reactions. The functional logic represents the target system functionality. In Shaw’s model, the monitored source is the system itself, and reactions are also communicated to the system, which forms a closed-loop. Shaw’s model is leveraged in Garlan et al. (2004) and in Cheng, Garlan, & Schmerl (2005) with the definition of “probes,” “gauges,” and “effectors” as a way to standardize connection interfaces between the adaptation logic and the functional logic, as shown in Figure 2.1. Probes and effectors should be integrated as part of the system functionality as an addition to the original system. Probes are interfaces to gather information from the system operation (changes). Gauges gather information from probes at run time. Effectors are interfaces to make adjustments altering system functionality.

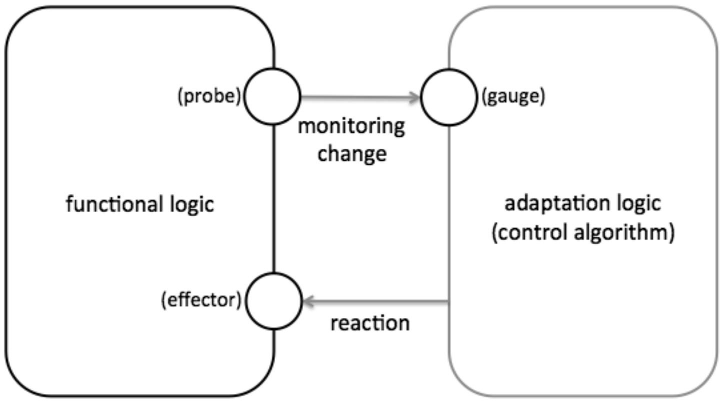


Figure 2.1. Closed-loop architectural model depicting functional logic, adaptation logic, and their interfaces.

Isolating the adaptation logic separates the concerns of system functionality from those of adaptation behaviors. With the adaptation mechanism as a separate entity, engineers can modify and extend it and reason about its adaptation logic with ease. Furthermore, the separation of concerns allows the application of this model even to legacy systems with inaccessible source code, assuming that the target system provides, or can be instrumented to provide, probes and effectors. Finally, providing external adaptation logic with generic but customizable mechanisms facilitates reuse across systems, reducing the cost of developing new self-adaptive systems.

2.2. Affect and Affective Computing

Computers are traditionally treated as logical and rational tools (Hussain & Calvo 2009).

The field of affective computing aspires to narrow the communicative gap between the emotional human and the emotionally challenged computer by developing computational systems and devices that can recognize, interpret, process, interact with, and simulate human affect (Picard 1997). Affect refers to emotions, feelings, and moods, such as surprise, tiredness, boredom, etc. (Hussain & Calvo 2009). Affect can be recognized from physiological signals (such as brainwaves, heart rate, and skin conductance), facial gestures, speech features, and postures. These constructs have notoriously noisy, murky, and fuzzy boundaries that are compounded with individual differences and contextual influences in experience and expression. Affective computing is an interdisciplinary field, a branch of artificial intelligence spanning computer science, physiology, and cognitive science (Hussain & Calvo 2009). Affective computing advocates the claim that having computers recognize and respond to affect is an essential part of the next generation of human-computer interfaces (Norman, Ortony, & Russell 2003; Picard 1997); computer self-adaptation driven by a user's affect is expected to result in more usable, useful, naturalistic, social, and enjoyable software systems. However, creating realistic (i.e., real-time, multimodal, cost-effective) affective computing applications is still a research field (Hussain & Calvo 2009) in which a number of obstacles need to be overcome including: improvement of affect recognition devices and algorithms, advancement of fusion methods, and reconciliation of emotional state models. The state-of-the-art of these topics is described in the following three paragraphs.

Devices and algorithms. Researchers have risen to the affect recognition challenge by developing a variety of novel sensors, algorithms, and models. These include galvanic skin conductance sensors, which measure arousal (Strauss et al. 2005); pressure and posture sensors, which attempt to detect frustration or interest by applying classification algorithms (Qi & Picard 2002; Mota & Picard 2003); brain-computer interfaces, which use brainwaves (electrical activity along the scalp produced by the firing of neurons) as an information source (Vallabhaneni, Wang, & He 2005); face-based affect recognition, which captures images of facial expressions and head movements (Kaliouby & Robinson 2005); and eye-tracking systems, which measure eye positions, eye movement, and pupil dilation, which have been reported to be related to the intensity continuum of affective stimulation (Janisse 1973). A more complete survey in affect detection research can be consulted in Jaimes & Sebe (2007) and Calvo & D'Mello (2010). Six devices and algorithms are used in this dissertation and a quick reference of them is presented as an introduction:

a) The Emotiv EPOC headset. It is an inexpensive wireless hardware device, which uses EEG technology to sense electrical activity in the brain. The device provides constructs for excitement, engagement, boredom, meditation, and frustration at a sampling rate of 8 Hz and 14 channels of raw brainwave data at a sampling rate of 128 Hz.
b) Tobii T60XL eye tracker. It provides data about a user's focus of attention and focus time while performing a task on a screen, including gaze points (x and y values) and pupil dilation, at a sampling rate of 60 Hz.
c) MindReader. It is an affect recognition software that infers emotions from facial expressions and head movements, in real time, providing constructs for agreeing, concentrating, disagreeing, interested, thinking, and unsureness at a sampling rate of

10 Hz.

d) A custom built skin conductance device, developed in collaboration with the Affective Computing group at MIT Media Lab. It measures the electrical conductance of the skin. Skin conductance varies with skin moisture level, which depends on the sweat glands, which in turn are controlled by the sympathetic and parasympathetic nervous systems. Skin conductance is an indicator of psychological or physiological arousal (frustration, excitement, etc.). The device reports values at a sampling rate of 2 Hz.

e) A custom built pressure-sensing device, developed based on a prototype designed by the Affective Computing group at the MIT Media Lab. It detects the amount of pressure that the user puts on a controller (such as a mouse or game controller). These values can be correlated with levels of frustration, as reported by Qi & Picard (2002). It has six sensors reporting the pressure value for each at a sampling rate of 6 Hz.

f) An custom built posture-sensing device. It is a low-cost, low-resolution pressure sensitive seat cushion and back pad with an incorporated accelerometer to measure elements of a user's posture and activity, developed in-house based on the experience of using a more expensive high-resolution unit from the MIT Media Lab Affective Computing group (Mota & Picard 2003). It measures pressure values in the back pad and the seat cushion (in the right, middle, and left zones) at a sampling rate of 6 Hz.

Fusion methods. The use of several channels through a multimodal approach makes it possible to recognize a broad range of affect as well as to improve the accuracy of the process. Fusing measurements coming from diverse channels requires an adequate selection of an appropriate fusion method. Fusion methods are classified into three levels (Vildjiounaite et al. 2009) according to the three-step schema described by Clay, Couture, & Nigay (2009): (1) lower level methods merge "signals" captured from the sensors (raw data) and only can be applied when the sensors' information is of the same type; (2) feature level methods analyze sensor signals, extract features, and combine those features; and (3) decision level methods fuse the interpretations (inferred affect) reported for all channels. Support vector machines (SVM), decision trees, Gaussian process classification, and hidden Markov models have been applied in the implementation of fusion methods (Vildjiounaite et al. 2009).

Emotional state models. Emotional state models describe affective states and their causal relationships. The adoption of an emotional state model is a requisite in multimodal affect integration to serve as a formalism to classify and interpret the diverse inputs (Gilroy, Cavazza, & Benayoun 2009). However, the classification of emotions is an active part of affective science in which experts struggle to reconcile competing models. Discussion on the different views of emotion has occurred over the years, and two viewpoints have strengthened: the discrete model and the continuous dimensional model. The *discrete model* assumes affective states are discrete values and only a finite number of values are possible. Although, researchers disagree on the exact number of distinct values. For instance, Ekman (1992) concluded that there are six basic emotions (happiness/joy, sadness, surprise, fear, disgust, anger), while Tomkins (1962) concluded that there are nine basic emotions (enjoyment/joy, interest/excitement, surprise/startle, anger/rage, disgust, dissmell, distress/anguish, fear/terror, and shame/humiliation). In this model, affect is seen to be universal to all humans, biologically fixed, and arising from evolutionary bases tending to be strong reactions to high-intensity stimuli. Affect is supposed to have functional signatures related to physiological signals. A major issue here is to define a consensus on the method by which affect can be determined. For instance, Ekman (1992) points to using facial expressions as a method to identify a number of affective states. A limitation of this model is that it focuses on strong emotions, but for analyzing interactions it may be advantageous to use a model that accommodates different ranges of any emotion and different combinations of emotions. The *continuous dimensional model* asserts that affective states are continuous values in one or more dimensions and conceptualizes affect by defining where it lies in the dimensional space. Russell (1980) proposed a 2D model that links pleasure with arousal. Pleasure measures how pleasing or displeasing an emotion may be (spanning from positive to negative), while arousal measures the intensity of the emotion (spanning from calmness to excitement). Mehrabian (1996) proposed a 3D model that measures affect response along three dimensions (pleasure, arousal, and dominance) as shown in Figure 2.2. The additional axis, dominance, represents the controlling nature of the emotion, i.e., its influence over others and the surroundings. For example, boredom and engagement are opposite states – boredom lies in the low pleasure, low arousal, and low dominance quadrant, while engagement is in the opposite quadrant. Since continuous values characterize the measurement of affect during interactions, this dissertation proposes to use Mehrabian's model (also known as the PAD model). An example of the use of the PAD model in real-time systems (for an art installation) is described by Gilroy, Cavazza, & Benayoun (2009).

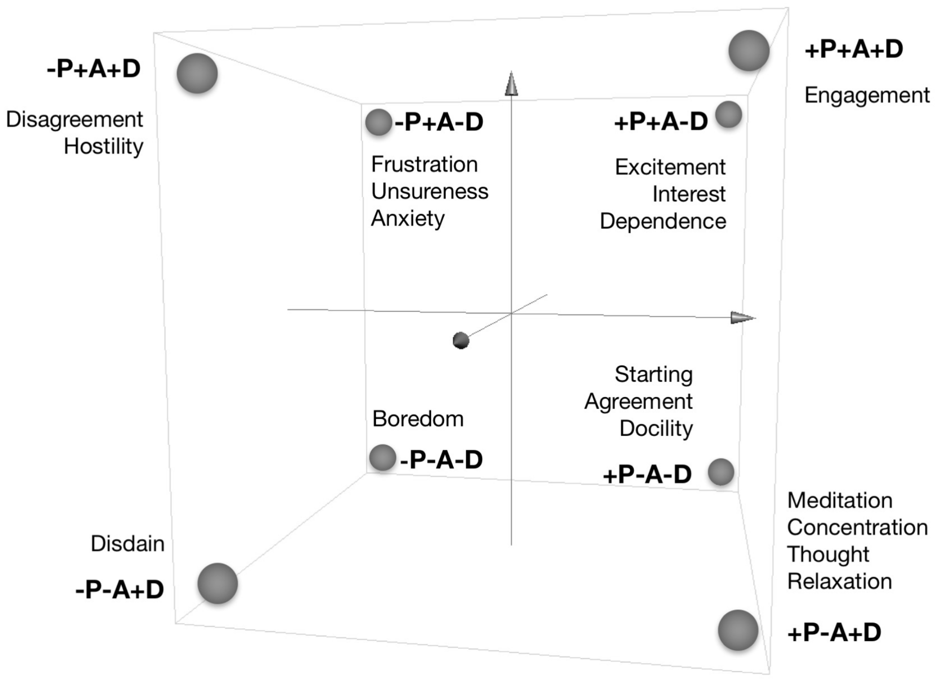


Figure 2.2. Pleasure-Arousal-Dominance dimensional model.

2.3. Software Product Lines

As the size and complexity of software systems increase, the design problem goes beyond algorithms, data structures (Feldman 2004), and a people-oriented discipline reliant on the craftsmanship of skilled individuals engaged in a labor-intensive manual task approach. Moving software construction to an engineering approach where design is done in a systematic, disciplined, and quantifiable way requires the application of software engineering approaches. One of those approaches, which supports a manufacturing vision, is the use of SPLs. SPLs capture and encode knowledge and proven practices to allow systematic development of systems that share a common set of features and satisfy the specific needs of a particular market segment or mission. SPLs are inspired by the proven benefits of product lines in manufacturing and create new functionality by integrating many ready-built and built-to-order components instead of designing and writing large amounts of new code in house from scratch. SPLs are supported by the advent of object and component technology; in particular, model-driven and component-based software design approaches are fundamental to the philosophy of SPL practices (Clements, Kazman, & Klein 2001).

The goal of SPLs is to save engineers time and development effort. SPLs take advantage of commonality and bounded variation through systematic reuse of design and implementation of successful software assets that have already been designed, developed, and tested for an application domain; therefore, SPLs yield predictable results (Clements, Kazman, & Klein 2001). Systematic reuse differs from opportunistic reuse (in which developers cut and paste code from existing programs to create new ones). Systematic reuse is an intentional and concerted effort to create and apply multi-use software artifacts throughout an organization. Systematic reuse increases software productivity and quality by breaking the costly cycle of rediscovering, reinventing, and revalidating common software artifacts (Schmidt & Buschmann 2003). Throughout the rest of this document when the term "reuse" is used, it means "systematic reuse." Reusing techniques involves code reuse, design reuse, or both.

Although SPLs have various benefits, it is important to establish that SPLs have a payoff-point where the cumulative cost of current development practices intersects with the cumulative cost of the SPL approach based on the number of products expected to be implemented (Weiss & Lai 1999; Clements, Kazman, & Klein 2001). If the goal was to implement just one or two affect-driven self-adaptive systems, SPLs would not be the best option, but as stated before, the goal is to broadly support developers to adopt these capabilities into their systems.

Defining an SPL involves three technical elements: an architecture, an infrastructure of components, and production guidelines. Each of these topics has essential background concepts that are described in the following paragraphs: (1) what an architecture is; (2) what components are and why they are integrated into frameworks; and (3) how patterns and pattern languages can be used for documenting guidelines.

Architecture. The core of the SPL's assets is formed by a software architecture, which denotes the overall structure shared by the set of systems. The architecture defines how systems must be built, advancing common problems and generalizing them while addressing the feasibility and quality attributes (Garlan & Shaw 1994).

Components. A component can range from a class, a set of classes, or a module that encapsulates code (both data structures and algorithms) to libraries or complete platforms. Components are built on the extensive paradigm of object-oriented programming and object-oriented design. SPLs define three categories of components: (1) core components that will be used for all the products, (2) optional components that may or may not be present in a product, and (3) adjustable components that might be tailored according to the product's needs. Components tend to be organized into frameworks and production guidelines defined to describe how to customize and combine them (Kastner, Apel, & Kuhlemann 2008).

Frameworks. Frameworks encapsulate components and describe interfaces that make it possible to mix and match these components to build a wide variety of systems from a small number of existing components. Frameworks provide a reusable context for components – a standard way for components to handle errors, to exchange data, and to invoke operations on each other (Johnson 1997). Frameworks are separate from normal libraries because they contain three key distinguishing features: inversion of control, default behavior, and extensibility (Schmidt & Buschmann 2003).

Patterns. Patterns are one of the more accepted approaches to document interconnection and usage rules of components (Johnson 1992); therefore, they are suitable to document production guidelines. A pattern is a template that consists of a name, an essay that describes a problem to be solved, a solution, the context in which the solution works, and a description of its costs and benefits. Patterns provide a common vocabulary for describing designs, provide a way to make design tradeoffs explicit, and promote creative freedom (Ramirez & Cheng 2010). Patterns address the problem of documenting the rationale of experienced designers within the code, providing developers with the means to escape traps and pitfalls that traditionally have been avoided only via long and costly apprenticeships (Johnson 1997; Schmidt & Buschmann 2003). A quick reference of design patterns and architecture patterns used in this work is presented as an introduction:

a) *Observer*. It defines a one-to-many dependency between objects so that when one object changes a value, all its dependents are notified and updated automatically (Gamma et al. 1995).
b) *Delegate*. It defines a one-to-many dependency between objects so that an object, instead of performing one of its stated tasks, delegates that task to an associated

helper object (Deugo 1998).

c) *Facade*. It provides a higher-level unified interface to a set of interfaces in a subsystem. It makes the subsystem easier to use (Gamma et al. 1995).

d) *Publisher-Subscriber*. It defines a one-to-many dependency between objects so that some objects, called publishers, produce and send message while other objects, called subscribers, which express interest in receiving messages, receive them. Publishers works without knowledge of which subscribers, if any, there may be (Buschmann et al. 1996).

e) *Blackboard*. It defines a one-to-many dependency between objects so that one object, called Blackboard, store and integrate large and diverse specialized information provided in an asynchronous way from other objects, called sources. The Blackboard also implements control strategies to decide which source will have access to post information. (Buschmann et al. 1996). As analogy, the Blackboard can be seen as a fusion of a board and a professor in a classroom while the sources are students participating in the solution of a problem.

Pattern Languages. Grouping patterns together and defining grammatical and semantic relationships between them creates a pattern language. A pattern language describes a solution space for developing a particular type of system by offering alternative solutions to common problems in a specific domain (Buschmann, Henney, & Schmidt 2007). A pattern language that records knowledge related to the composition and organization of affect-driven self-adaptive systems, an extension of the work described in Gonzalez-Sanchez, Chavez-Echeagaray, Atkinson & Burleson (2012), guides the description of the

ADASPL domain and the definition of the ADASPL architecture presented in Chapter 3.

2.4. Process Measures and Software Metrics

To quantify the SPL development process and the properties of the software product produced, three kinds of metrics were collected for each project:

a) *Process measures*. These quantify the involvement of people in the project as they add, remove or update files over time. File updates involve the adding or removing of lines. Process measures are extracted with GitStats,[1] a statistics generator for Git-distributed revision-control system repositories. GitStats examines the repository and produces statistics based on the history of the project. Git repositories were provided for each project and their use was enforced. Specifically, GitHub, a web-based Git repository hosting service was used. Git repositories store all the assets for the project including code and documentation. It is important to note that lines (in commits) are not the same as lines of code (referred as a metric), because diverse artifacts (files), not only code, are stored in the repository.

b) *Software complexity metrics*. These quantify the complexity of the software measuring its size (number of files, methods, and LOC), its uses of inheritance, and its cyclomatic complexity. These metrics are extracted with a quality analysis tool called RSM (Resource Standard Metrics)[2].

c) *Software structure metrics*. These quantify the design flaws of the software by measuring cohesion and coupling. These metrics are extracted with a structure

[1] http://gitstats.sourceforge.net
[2] http://msquaredtechnologies.com/rsm-wizard.html

analysis tool called stan4j (Structure Analysis for Java)[3].

In the next sections process measures and software metrics used are described. The software metrics and their value ranges are summarized in Tables 2.1, 2.2, and 2.3, respectively.

Table 2.1. Summary of Complexity Metrics

Name	Suggested range
LOC	▪ 200 per method ▪ 1000 per class
DIT	▪ Less than 2 is poor use of OO ▪ 2 to 5 is acceptable ▪ More than 5 is complex
NOC	▪ 0 to 10 is fine ▪ More than 10 is complex
CC	▪ 0 to 10 is acceptable ▪ 10 to 20 is complex ▪ More than 20 could be a problem

Table 2.2. Summary of Cohesion Metrics

Name	Suggested range
FAT	▪ Less than 60 is recommended ▪ 60 to 120 is acceptable ▪ More than 120 is not recommended
H	▪ 1.5 to 4.0 is recommended

Table 2.3. Summary of Coupling Metrics

Name	Suggested range
I	▪ 0 is stable, 1 is unstable
A	▪ 0 is concrete, 1 is abstract
D	▪ Less than 0.1 is recommended ▪ Less than 0.4 is acceptable ▪ Zone where A = 0 to I = 0 is a pain zone ▪ Zone where A = 1 to I = 1 is a useless zone

[3] http://stan4j.com

2.4.1. Process Measures

The software development process is quantified by measuring the following:

a) Number of commits. Committing is a process that adds, modifies, or removes files from the repository (source files or documents). This is quantified per author and per date.

b) Number of lines. Lines added and removed, including lines of source code (LOC) and text in documents. This is quantified per author and per date.

2.4.2. Software Complexity

Software complexity is quantified by measuring software size (quantifying files, methods, and LOC), object-oriented design (quantifying DIT and NOC), and cyclomatic complexity (Sommerville 2002; Chidamber & Kemerer, 1994; McCabe, 1976). These metrics are as follows:

a) Files in the project.

b) Methods. The total number of functions within the source code determines the degree of system modularity.

c) LOC. This counts the lines but excludes empty lines and comments. This metric represents more accurately the quantity of work performed. An accepted industry standard is 200 LOC per function or 1,000 LOC per class. Functions that have a higher LOC are difficult to comprehend and maintain. XML files, configuration files and auto generated files were excluded in the calculation of this metric.

d) Average depth of inheritance tree (DIT). This calculates how far down a class is declared in the inheritance hierarchy, where the DIT is the length from the class to the root of the inheritance tree. The deeper a class is in the hierarchy, the greater the number of methods and state variables it is likely to inherit, which makes it more difficult to predict its behavior. It becomes more specialized and it can be hard to understand a system with many inheritance layers. However, there is greater potential reuse of inherited methods. If a majority of DIT values are below 2, this may represent poor exploitation of the advantages of OO design and inheritance; however, the recommended DIT is 5 or less.

e) Average number of children (NOC). This measures the number of direct subclasses of each class. The NOC approximately indicates how an application reuses itself. It is assumed that the more children a class has, the more responsibility there is on the maintainer of the class not to break the children's behavior. The upper recommended NOC limit for a class is 10 and the lower limit is 0. If the NOC for a class exceeds 10, this may indicate a misuse of sub-classing.

f) Cyclomatic complexity. This is the degree of logical branching within a function. Logical branching occurs when “while”, “for”, “if”, or “case” keywords appear within the function. It was calculated the total, average, minimum, and maximum value for this metric. Cyclomatic complexity is the count of these constructs. It is commonly accepted that a cyclomatic complexity of between 1 and 10 is considered simple and easy to understand, between 10 and 20 indicates more complex code, which may still be comprehensible, and values of 20 and above are typical of code with a very large number of potential execution paths that can only be fully grasped and tested with great difficulty and effort.

2.4.3. Software Structure

Software structure quality is quantified by measuring cohesion and coupling at the package level using the metrics proposed by Robert C. Martin (2013).

Cohesion, the grouping of classes, is measured by two metrics: FAT of dependencies among a package's classes and relational cohesion.

a) Fat for top level class dependencies (FAT) measures the dependency among classes forming the package and is calculated as the edge count of the package's class dependency graph. Values less than 60 are encouraged, values less than 120 are acceptable, and values above 120 represent a warning.

b) Relational cohesion (H) represents the average number of internal relationships per class in a package. As classes inside packages should be strongly related, the cohesion should be high. However, high values may indicate over-coupling. It is calculated as $H = (FAT + 1) / \text{number of files}$. A good range of H is 1.5 to 4.0. Projects where $H < 1.5$ or $H > 4.0$ might be problematic. This measure is less applicable to packages consisting mostly of interfaces, i.e., it is useful for packages that contain implementation classes.

Coupling, or the relationships between packages, is measured using three metrics: instability, abstractness, and distance

a) Instability (I) is related to the amount of work required to make a change. A package is considered stable if it is difficult to change, and it is considered difficult to change if has lots of others packages dependent on it, i.e., it has a lot of incoming dependencies. On the other hand, a package is considered unstable if it has few or no incoming dependencies. To measure instability, the number of dependencies that enter and leave a package are counted.

Two values are important to calculate: (1) afferent coupling and (2) efferent coupling. Afferent coupling (Ca) is the number of classes outside the package that depend on classes within the package. Efferent coupling (Ce) is the number of classes inside the package that depend on classes outside the package. I is defined as the ratio of Ce to total coupling (Ce + Ca), such that I = Ce / (Ce + Ca). The range for this metric is 0 to 1, with I = 0 indicating a completely stable package and I=1 indicating a completely unstable package.

b) Abstractness (A) measures the abstraction level of the package. A is the ratio of the number of abstract classes (including interfaces) in the package to the total number of classes in the package. The range for this metric is 0 to 1, with A = 0 indicating a completely concrete package and A = 1 indicating a completely abstract package.

c) Distance (D) is the perpendicular distance of a package from the idealized main sequence line given by A + I = 1. It is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract and stable (I = 0, A = 1) or completely concrete and unstable (I = 1, A = 0). D is calculated as D = | A + I - 1 |. The range for this metric is 0 to 1, with D = 0 indicating a package that is coincident with the main sequence and D = 1 indicating a package that is as far from the main sequence as possible. A maximum D of 0.1 is recommended. A value for D of less than 0.4 is acceptable. The zone where

A = 1 and I = 1 is useless and the zone where A = 0 and I = 0 suggest a rigid and undesirable package that is not extensible and difficult to change.

2.5. Related Work Addressing Affect-Driven Self-Adaptation

This section summarizes milestones that have occurred for a broader deployment of the affect-driven self-adaptation capability to software systems. In short, capabilities required for building adaptation based on human factors, such as affect measurements, have not been considered and most research does not have a manufacturing vision or does not follow (or does not report to follow) software engineering models or guidelines. I describe various threads of related work, presenting their similarities and differences with the approach followed in this dissertation and highlighting advantages and limitations. Each subsection below addresses a different project and a summary is provided in the last section.

A Platform for Affective Agent Research. Burleson et al. developed an Affective Learning Companion (ALC) platform for multimodal sensing and interpretation of affective information, which is able to provide a response in real-time through an expressive agent (Burleson et al. 2004). Details and extensions to the platform are documented in Kapoor & Picard (2005) and Kapoor, Burleson, & Picard (2007). The ALC platform is formed by six modules (sensing, inference engine, behavior modification engine, server, logger, and the animated character engine), which work as follows: first, multiple hardware channels (including eye tracking, pressure sensors, face-based emotion recognition, and skin conductance) sense user changes and report their measures to a server; then, the server continuously stores the data in a text file and runs an inference engine process that uses this file as input for a semi-supervised machine-learning inference process; the inference engine uses training data to identify specific affective states such as frustration, interest, or boredom; finally, the identified affect state is used as the parameter that configures the behavior of an animated character; and the behavior is chosen from a set of predefined scripts.

A Middleware for Self-Adaptation. Garlan et al. (2004) recognize that adaptation mechanisms highly specific to one application and tightly bound to the code are costly to build and difficult to modify; therefore, they proposed a middleware for supporting adaptation. The middleware leverages a framework, called Rainbow, and guidelines to work with it. Their goal is to be able to add self-adaptation capabilities to a wide variety of systems, adding external control mechanisms and using a reusable infrastructure.

Human-in-the-loop adaptation. An extension of Garlan et al.'s (2004) research is presented by Camara, Moreno & Garlan (2015), who research how to involve humans as information sources, decision-makers, or system-level effectors (executors of the adaptation when automation is not possible or as a fallback mechanism). They researched the explicit modeling of humans to provide a better insight into the trade-offs of involving humans in the adaptation loop for safety-critical systems; particularly, considering that humans are influenced by factors external to the system (e.g., training level, fatigue). Their goal is to combine human interventions and automated adaptations.

An Architecture Pattern for Affect Awareness. Clay, Couture, & Nigay (2009) proposed an architecture pattern, named emotion recognition branch, as a guide for engineering affective systems. Clay, Couture, & Nigay's architecture pattern is formed by three piped components for capturing, analyzing, and interpreting data, as follows: (1) the capture unit groups sensor interfaces to acquire raw data from sensors; (2) the analysis unit extracts affective relevant cues (features) from the captured data; and (3) the interpretation unit is dedicated to the interpretation of cues (features) to infer the affective state. Additionally, it defines filters, adaptors, and concentrators, which transform the data flow, format, and merge flows together, respectively. A case study is described without technical details, inferring affect from a dancer's body gestures using a motion capture system as input source.

A Framework for Affect Recognition. Hussain & Calvo (2009) proposed a framework for multimodal affect recognition targeted for learning systems but flexible to other applications. The framework does multimodal affect recognition and is able to report results to third-party systems by sharing an XML file with them. The Hussain & Calvo framework defines a set of four modules for feature extraction, feature selection and reduction, feature classification, and decision fusion. Features from multiple modalities (including physiological signals, speech analysis, and text analysis) are handled and fused with decision-level techniques. Modules are implemented in MATLAB and connected with batch processing using XML files as connectors between modules.

Affective and Adaptive User Interface in Office Scenarios. Maat & Pantic (2007) describe a system, named Gaze-X, developed to support affective multimodal human-computer interaction in which a user's actions and affect are used to adapt user interfaces. Multiple hardware channels (including eye tracking, speech, and face-based emotion recognition) sense changes in user affect, and the user's activity (keystrokes and mouse movements) is tracked while running applications (web browser, mail, text editor, Adobe reader, and music or video player). Adaptation strategies include help provision, addition or removal of automation tasks, and changing information presentation.

A Framework for Smart Sensor Integration. Wagner, André, & Jung (2009) describe a framework that supports the development of multimodal emotion recognition in real-time named Smart Sensor Integration (SSI). SSI offers tailored tools for data segmentation, feature extraction, and pattern recognition, both to be applied offline (training phase) and online (real-time). SSI is a three-layered framework that is able to handle inputs from various input modalities and uses the Pleasure-Arousal-Dominance (PAD) model for fusion measurements and to share results with third-party systems. It provides an infrastructure to include function calls to external resources, such as signal processing libraries, machine-learning libraries, and classifier algorithms; it integrates several libraries such as OpenCV, ARToolKitPlus (real-time marker tracking), SHORE (face detection), EmoVoice, and Torch (machine learning). Additionally, it offers a generic GUI for data acquisition and training. SSI has been used to develop story telling interactive systems (with real-time vocal emotion recognition), augmented reality art installation (using a speech recognizer and a Wiimote to alter a virtual tree), and a virtual butler that responds to the user's affective state.

A Framework for Audio-Visual Affect Recognition in a Crowd. Vildjiounaite et al. (2009) describe a framework for multimodal audio-visual affect recognition in a crowd. The framework was validated in three contexts: theatre, circus, and sports events with the audience waiting, leaving, and during the show (looking for approval or interest); the shots of the audience were obtained from movies and TV programs. This framework incorporates diverse approaches for fusion methods and points to the importance of context-awareness during the affect recognition process. It is a modular framework with a complex component-based architecture model, although no mention is made or evidence shown about the use of patterns or styles. It supports several machine-learning algorithms. Additionally, it supports training and classification phases for the creation of user models.

A Reference Model for Self-adaptation. Weyns, Malek, & Andersson (2010) defined a reference model in which one or more independent self-adaptive units collaborate to achieve self-adaptation. Case studies were conducted on intelligent transportation systems, using cameras equipped with data processing and communication capabilities to monitor traffic jams. The model of Weyns et al. is an agent-based system organized in four layers: host infrastructure, agent middleware, organization middleware, and agents layer. A technical feature relevant in this approach is that it brings together the concepts of computational reflection and controlled-loop and extends it with constructs necessary for decentralization.

Other Efforts. Several other efforts exist, most of them similar to one of the previously described and thus with similar limitations. Some of these efforts are as follows: multimodal affective user interface by Lissetti & Nasoz (2002); agent-based intelligent tutoring system using a facial recognition system by Gowri et al. (2010); a framework for an affective intelligent tutor system (Emilie-1 and Emile-2) by Nkambou (2006); the SEMAINE framework by Schroder (2010); and multimodal emotion recognition by Sebe et al. (2005). Moreover, in the field of context-aware systems, Hong, Suh, & Kim (2009) present a literature review and conclude that social science methodologies, psychology, cognitive science, and human behavior related approaches are required and can be implemented in context-aware computing.

Table 2.4 presents a summary of the related work described in this section including the key elements on affect, adaptation, and software present or absent in each related work. The first column provides the reference to the work in the same order that they were previously presented. The second column states whether the work is about a system (one application), a framework or middleware (components able to be reused), or a model (such as a pattern). The third column names the system developed or states whether a generic approach was followed, i.e., several systems can be created. The fourth column identifies whether the work involves at least the concept of affect. The fifth column identifies whether this work supports adaptation. The sixth column states whether or not this work focuses on software developers. Finally, the last column records the architectural style reported.

Table 2.4. Comparison of Related Work

Reference	Type	Use	Affect	Adaptation	User or developer oriented?	Architecture
Burleson et al. 2004	system	animated agent	■	closed loop	user	Batch
Garlan et al. 2004	middleware	generic		closed loop	developer	Layered
Camara et al. 2015	middleware	generic		closed loop	developer	Layered
Clay et al. 2009	pattern	generic	■		developer	Pipes & Filters
Hussain and Calvo 2009	framework	generic	■		user	Batch
Maat and Pantic 2007	system	adaptive UI	■	closed loop	user	Agents
Wagner et al. 2009	framework	generic	■	closed loop	developer	Layered
Vildjiounaite et al. 2009	framework	affect in crowds	■		user	Not specified
Weyns et al. 2010	framework	generic		distributed	developer	Layered

Three topics were the focus of the survey of previous work (manufacturing vision, self-adaptation, and affect). In relation to software engineering, the approaches move from creating a one-of-a-kind system to a manufacturing approach; being in the middle represents that the approach supports developers but not broad deployment. Related to self-adaptation, works reviewed move from being aware of something to adapting to that something. Finally, related to affect, the scale was set up as binary values: the work mentions affect at all, yes or no.

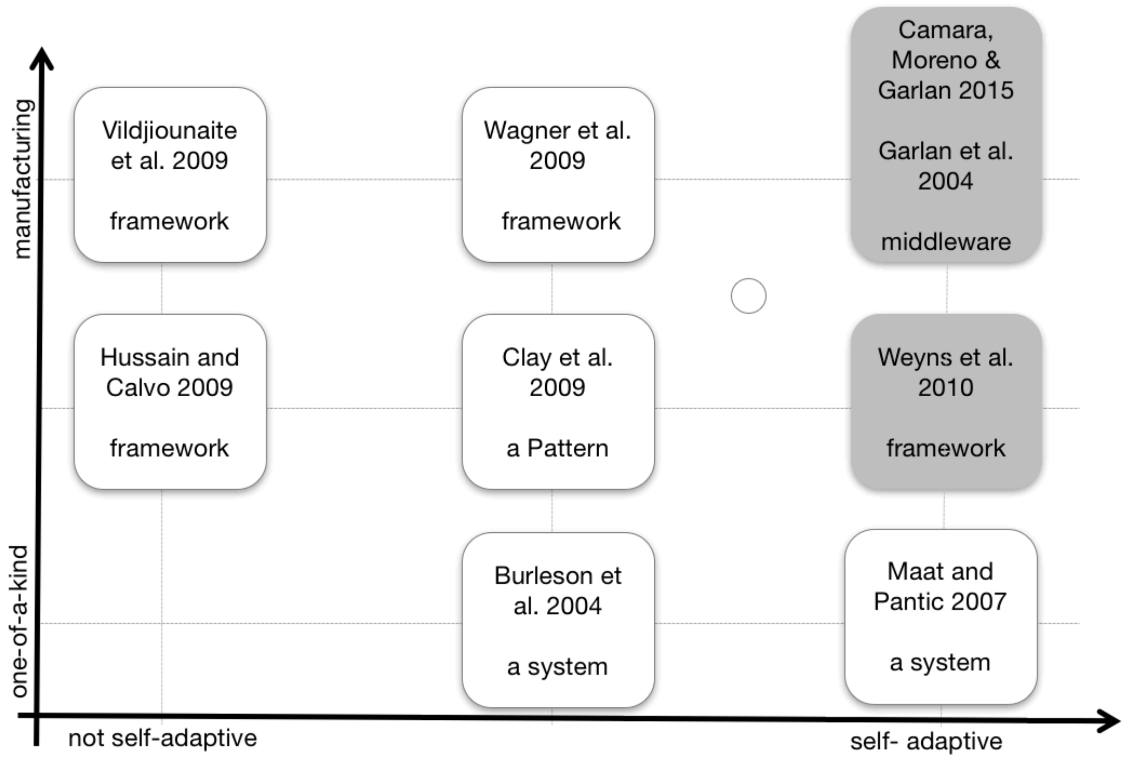


Figure 2.3. Graphical representation of the related work organized in a dimensional space in which the X-axis represents their focus on adaptation, the Y-axis represents their support for developers, and colors (white or gray) represent whether or not the project is related to affect recognition.

Figure 2.3 depicts the state-of-the-art in which the X-axis represents the degree a system is related to self-adaptation; the first group does not apply adaptation at all, the second group applies adaptation visualizing or changing a visual stimulus, such as an avatar behavior or art installation, and the third group is a next step of adaptation, a step in which adaptation rules can be defined and any aspect of the system functionality altered. The Y-axis represents the degree a system is related to a one-of-a-kind or a manufacturing approach, moving down to a means of no support at all for developers, some support, or a developer-oriented approach. Finally, dark gray boxes represent works not related to affect and white boxes represent works related to affect. As noted, affect is present in almost all related work except in those approaches that look forward to applying a manufacturing vision to self-adaptation. The work presented here intends to position itself in the upper-right corner of the space.

2.6. Limitations to State-of-the-Art Addressed

Limitations of the state-of-the-art includes:

The ALC platform: (1) it responds in real time, aligned with the idea of self-adaptation; (2) it identifies and states the significance of multimodality; and (3) it divides the responsibilities into modules. It shows that affect can be used to drive adaptation. Nevertheless, the platform of Burleson et al. (2004) aims to help end-users (researchers) run investigations and test theories about affective responses in learning; thus, it differs from this dissertation in its scope which is not an intentional and concerted effort to create and apply multi-use software artifacts to support developers in the creation of affect-driven adaptation; however, opportunistic reuse (cutting and pasting code from existing programs to create new ones) can be applied.

The Garlan et al. (2004) middleware: (1) it applies the closed-loop paradigm; (2) it defines a layered architecture that separates inference, adaptation, and rule evaluation; (3) it provides a framework to facilitate the creation of families of systems by means of components, constraints, properties, and qualities; and (4) it models interfaces for gathering information (probes), interfaces for carrying out a system modification (effectors), and interfaces for constraining evaluation (gauges). However, the Garlan et al.

middleware differs from this dissertation in the variable that drives the adaptation. It does not have the scope of dealing with affect measurements. Garlan et al.'s case studies address self-adaptation driven by performance, work balance, and other resource-related issues. Dealing with affect measurements increases the complexity of the problem and requires additional layers, new engines, and guidelines.

The human-in-the-loop adaptation approach from Camara, Moreno & Garlan (2015) is built upon the Garlan et al. middleware. Humans are included in the adaptation loop by considering them effectors (sophisticated sensors). Therefore, humans are used as another information source in the decision process. Human involvement is modeled to capture factors that can disturb human behavior and its interactions with the system, such as training level and fatigue. Their work is particularly focused on safety-critical systems, where the goal is to keep the system performance as expected to avoid death or serious injury to people, or loss of or severe damage to equipment and property.

The work of Clay, Couture, & Nigay (2009): (1) it is a generic architecture level abstraction that defines modules conceived to be integrated to the target system architecture; (2) it is based on the grounding theory of multimodality; and (3) it reflects the qualities of modifiability and reusability. Nevertheless, the work of Clay, Couture, & Nigay focuses only on the architecture definition and does not provide support to developers related to code or components to be applied; Clay, Couture, & Nigay mention that as future work. Moreover, the work of Clay, Couture, & Nigay is dedicated to engineering affect-aware systems; what the system does with the information about the affect measurements is not supported. In my dissertation affect recognition is not a goal itself, but rather a means to achieve self-adaptation.

The Hussain & Calvo (2009) framework: (1) it identifies and states the significance of multimodality and (2) it divides the responsibilities into modules. Nevertheless, the Hussain and Calvo framework does not have an intentional and concerted effort to create and apply multi-use software artifacts to specifically support developers in the creation of affect-driven adaptation. It is mostly a platform similar to the one described in Burleson et al. (2004) to support end-users (researchers) in running investigations and specifically in testing theories about affective responses in learning; I argue that this is not a framework in the software context defined by Johnson (1997) to help developers create software. No mention is made about real-time capabilities and thus about supporting self-adaptation.

Technical features in Maat & Pantic's work (2007): (1) it implements a real-time closed-loop; (2) it adds to the process context-sensitivity identifying user activity and mapping it to the sensed emotion; (3) it is a system that runs independently of the applications that the user is running; (4) it implements both supervised and unsupervised learning to gather user preferences; and (5) it identifies and states the significance of multimodality and focus on adaptation (i.e., real-time processing) not only in affect recognition. However, the Maat & Pantic's system appears to be helpful for end-users (researchers) for testing adaptability in user interfaces; there is no intentional and concerted effort to create and apply multi-use software artifacts to support developers to create affect-driven adaptation. The system as described can be used as a whole for affect recognition; this is also noticed and stated by Wagner et al. (2009). Maat & Pantic use a framework, called Fleebe, to implement their system to run studies in office environments. While Fleebe is a framework, the modules related to affect (which are outside the Fleebe framework) only provide opportunistic reuse.

The SSI framework created by Wagner, André, & Jung (2009): (1) its target audience is developers and it has an intentional and concerted effort to create and apply multi-use software artifacts where encapsulation, systematic reuse, delegation, and decoupling are widely supported and suggested; (2) it provides assets to let developers choose which aspects to monitor, which conditions trigger adaptation, and how to adapt the system; and (3) the PAD emotional state model is used for fusion measurements. Even though the framework supports real-time affect recognition and enables adaptation, the framework itself does not provide models or guides about the self-adaptation part of the system. Still, the accomplishment of achievements similar to the one described in this project is an inspiration.

Vildjiounaite et al. work (2009): (1) the PAD emotional state model is used for fusion measurements; (2) it identifies and states the significance of multimodality; and (3) it follows a modular approach. Nevertheless, this platform is similar to the one described in Burleson et al. (2004) and Hussein & Calvo (2009) – it can be used as a whole for multimodal affect recognition and can report results to third-party systems – but I argue that this is not a framework in the software context defined by Johnson (1997) as "a skeleton for an application that can be customized" or "a set of reusable components and the guidelines for their interaction." Also, its scope is not an intentional and concerted effort to create and apply multi-use software artifacts to support developers in the creation of affect-driven adaptation but to help end-users (researchers) facilitate research on affect recognition.

Weyns et al. (2010) work, as the one I am proposing, describes a systematic engineering approach that observes trade-offs among design decisions and addresses software qualities. However, the research of Weyns et al. does not address self-adaptation driven by affect measurements or the issues related to it.

In summary, existing approaches only partially tackle the problem of deploying affect-driven self-adaptive capabilities to software systems; some of them limit their applicability to supporting the craftsmanship of one product instead of a common product platform to manufacture product families, while others are focused on manufacturing self-adaptive but not affect-driven systems. This dissertation addresses two key issues not addressed by the above efforts: (1) modeling the process of self-adaptation driven by affect measurements with the aim of making it understandable to the system owners and developers, and (2) the issue of manufacturing limitations.

2.7. Foundational Elements Provided by the State-of-the-Art

I argue, in the previous section, that the state-of-the-art of affect-driven self-adaptation does not provide software engineers with a generalized, feasible, and cost-effective approach for incorporating the affect-driven self-adaptation capability into software systems. However, it does provide the foundational elements necessary to do so, as follows:

a) advances in affective computing provide the affect recognition devices, affect models, and affect inference techniques that allow computers to recognize, represent, and learn about user affective states;

b) advances in adaptive systems provide the models and mechanisms to enable the implementation of self-adaptation capabilities; these separate the adaptation process and the adaptation concerns from the target system's functionality; and
c) research in software engineering provides options for implementing a manufacturing vision for the development of these software systems.

An SPL paradigm is employed to provide the missing support and accomplish the goal of a generalized, feasible, and cost-effective approach to manufacturing software systems with the affect-driven self-adaptation capability. The SPL approach was selected as a manufacturing strategy to take advantage of its ability to encapsulate existing knowledge (such as advances in affective computing and adaptive systems research) into assets for software system design and development.

As stated in Chapter 1, although SPL covers both the technical and organizational aspects, this research takes advantage of technical solution strategies only and does not engage with the organizational specifics.

SPL defines its technical solution strategy as an incremental process for engineering products based on an architecture, an infrastructure of components, and a set of process guidelines. The purpose of each of these elements is described in Figure 1.1. and summarized below:

a) the architecture supports generality and feasibility by providing a basis for understanding and analyzing common behaviors in a family of products (in this case affect-driven self-adaptive systems) and supporting early design and implementation decisions, as well as facilitating communication between stakeholders;
b) the infrastructure supports generality and cost-effectiveness by providing a well-

understood set of components (in this case for affect recognition and adaptation); and

c) the process guidelines support feasibility and cost-effectiveness by helping engineers to understand the details of particular affect and adaptation concepts as operational capabilities, and by supporting their implementation in the use or extension of the provided infrastructure of components.

Each of the elements is described in a separate chapter. Each chapter explains how the individual element addresses particular challenges. Chapter 3 focuses on the architecture, Chapters 4 and 5 on the infrastructure, and Chapter 6 on the process guidelines. Furthermore, Chapter 6 also presents and discusses working examples overviewing the production process.

2.8. Summary

Current advances in adaptive technology provide mechanisms to enable self-adaptation, in particular to monitor events inside the target system and effect changes, which form building blocks to advance system control. Advances in affective computing research provide affect recognition devices and algorithms, fusion methods, and emotional state models. Advances in software engineering provide rigorous partitioning and composition of components techniques, abstraction and modeling methodologies, and recently approaches that adopted a manufacturing vision to change methods that were inherently focused on building one product at a time to the realization of families; in particular, software product lines which promise greater gains and open the door to mass deployment. Related research in affect-aware and affect- driven adaptation demonstrates point solutions as shown and described in Figure 2.3., which presents a number of limitations and unresolved issues that are addressed in this dissertation. Almost no existing approach provides a systematic, integrated approach that combines affect, adaptation, and software engineering. While existing approaches do not address system-level details that engineers grapple with in order to build their systems, I design an SPL approach, with a model, an infrastructure and guideline elements, that leverages a global perspective. The infrastructure encapsulates core concepts and hoists them as building blocks for systems engineering to build the affect-driven self-adaptation capability. The overall approach for affect-driven self-adaptation is presented in Chapters 3 to 6.

CHAPTER 3
DOMAIN ABSTRACTION AND ARCHITECTURE DEFINITION

ADASPL encapsulates existing advances in affective computing and adaptive systems research into assets for software system design and development. It defines a technical solution strategy as an incremental process for engineering products based on an architecture, an infrastructure of components, and a set of process guidelines. This chapter focuses on the definition of the first element, the architecture, and how it overcomes the limitations outlined in Chapter 2, addressing aspects of generality and feasibility. However, the definition of an architecture requires knowledge about the domain, e.g., the meaningful concepts, process steps, and data involved. Hence, this chapter starts by describing these elements of the domain.

3.1. Domain Abstraction

Meaningful concepts, process steps, and the data involved in affect-driven self-adaptive systems are documented in a pattern language, where each pattern documents an identified step in the process of affect-driven self-adaptation. This pattern language is an extension of the work described in Gonzalez-Sanchez, Chavez-Echeagaray, Atkinson & Burleson (2012). It includes seven interrelated patterns: sensing, perception, integration, synapse, reaction engine, introspection, and behavior management. These patterns were harvested in a pattern mining process examining existing systems. The patterns and their relationships are visualized in Figure 3.1 and are as follows:

a) *Sensing* is about establishing a connection with a hardware device and collecting the signal readings (also called raw data) that the device gathers, for instance, electrophysiological measurements, facial gestures, posture, or brainwaves. Numerous hardware devices can be used at the same time to collect signals from diverse sources, and each device should be attached to an individual sensing component. Each hardware device has particular characteristics, such as data polling rate and expected value ranges.

b) *Perception* is about interpreting the signal readings and inferring an affective state value. The input for perception comes from the output of sensing. For instance, joy can be inferred from facial gestures, interest can be inferred from posture while sitting in a chair, or meditation can be inferred from brainwaves. Sensing is a functionality that does not require changes after implementation unless the hardware or its communication protocol is changed; however, inference algorithm changes are commonly made to improve or refine the perception process. As a result, it is advisable to maintain the independence of each of these functionalities. While not all sensing processes require a perception process, all perception processes are dependent upon a sensing process. In some cases, sensed values are directly proportional to an increase or decrease in an affect measurement. In other cases, sensed values are already processed by the device platform and the signal readings include raw and inferred values. For instance, the Emotiv EEG device provides raw EEG data (14 channels) and five affect measurements. In these cases, the perception process can be null. But, most of the time, values gathered in the sensing process are not useful without an inference algorithm, provided by a perception process, to interpret them. For instance, brainwave, pressure, and posture readings need inference algorithms to interpret them.

c) *Integration* collects the values gathered by the sensing-perception pairs and then: (1) synchronizes them; (2) converts them to a predefined and normalized measurement unit; and (3) fuses them into one value. For instance, when using the Mehrabian emotional state model, the measurement unit is a 3D vector with values representing pleasure, arousal, and dominance and ranging from 0 to 1; a fusion approach that can be used with this model consists of adding the vectors together to produce a single vector value.

d) *Synapse* communicates the affective information to instances outside the affect recognition logic.

e) *Reaction engine* chooses and executes adaptation rules according to the inputs received from the synapse and introspection.

f) *Introspection* gathers information regarding the system's context (system status, responses, and failures) and UI events from the target system. For instance, in a tutoring system, a student clicking a button asking for hints to solve a problem or, in a 3D game, a player moving around and exploring the game scenario.

g) *Behavior management* stores and manages the adaptation rules that are executed by the reaction engine.

The sensing, perception, integration, and synapse steps fullfil the task of affect recognition, while the adaptation, introspection, and behavior management fulfil self-adaptation. The backbone of affect-driven self-adaptation is formed by the sensing, integration, synapse, and reaction engine steps, while perception, introspection, and behavior management are support steps. A system can use a sensing (and optionally a perception) step to acquire a monomodal affect recognition capability. Or, it can include several sensing (and optionally perception) steps, plus the integration and the synapse steps to acquire a multimodal affect recognition capability.

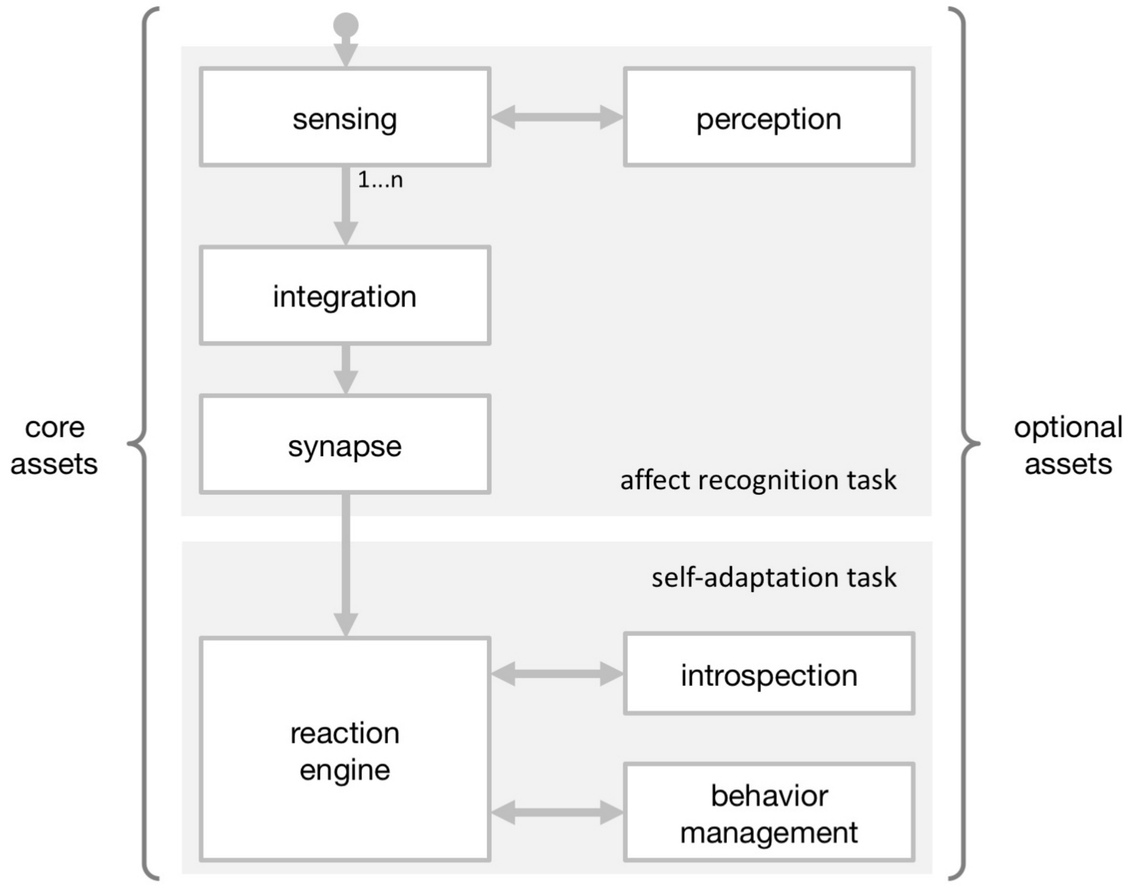


Figure 3.1. Patterns in the pattern language for affect-driven self-adaptive systems.

This domain abstraction drives the definition of the ADASPL architecture. An architecture definition is a recursive decomposition process where, at each stage, models and patterns are chosen to satisfy quality attributes and then functionality is allocated as described by the models and patterns selected. At the end, the architecture is described as a set of containers for functionality and the interactions among them. The following sections describe the sequence involved in the creation of the ADASPL architecture across multiple stages, from an overall model to three independent logics to the components contained in each logic.

3.2. Closed-Loop Model

The first challenge presented by human-centered systems, such as affect-driven self-adaptive systems, is how to include the user in the control loop. In order to tackle this, I extended the closed-loop adaptation model (Shaw 1995; Cheng, Garlan, & Schmerl 2005), described in Chapter 2, by integrating the human factor in the form of affect measurement. The proposed extension of the closed-loop adaptation model is shown in Figure 3.2. It maintains the respective independence of the functional logic and the adaptation logic, and incorporates a new, similarly independent element, an affect recognition logic. The interconnection between the three logics is standardized, maintaining the foundation for a model with low coupling and high cohesion. As in Cheng, Garlan, & Schmerl (2005), interfaces either extract information (gauge and probe pairs) or change the target system (effectors). The logics and their interactions can be summarized as follows:

a) The *affect recognition logic* monitors the user's affective state; it encapsulates affect recognition knowledge (which includes gathering the user's signals, processing them, and inferring affect), and makes its findings available to other logics and systems via the relevant probe.
b) The *adaptation logic* encapsulates the definition and execution of adaptation rules. It triggers changes in the functional logic using effector interfaces. The affective state reports from the affect recognition logic are used to trigger changes in the functional

logic. Adaptation tends to improve user experience and in doing so also can influence user affect.

c) The *functional logic* represents the target system, one to be newly implemented or an existing one. Changes triggered in the functional logic are intended to improve its functionality and, in doing so, the user experience, and consequently the user's affective state.

Affect-driven self-adaptation is improved by supplying information regarding the context of the affect, i.e., knowing what is happening inside the system (system status, responses, or failures) while the user is interacting with it and what inputs the user is giving to the system (UI events). For instance, in a tutoring system, the detection of frustration or concentration could mean different things depending on whether the user is reading or answering a quiz, or if the system is at a standstill. Adding system context awareness to the model requires adding a probe point to the functional logic and a gauge point to the adaptation logic. Now the adaptation logic is connected to the functional logic (as in Shaw's closed-loop model) as well as to the affect recognition logic. Figure 3.3 shows the model with this addition, in which the adaptation logic and functional logic characterizations are expanded as follows:

a) The *adaptation logic* establishes a closed-loop control that monitors both the user affective state and the system events, processes the information gathered, and then triggers adaptive reactions in the system. Affective states and system events are acquired by the gauges connected to the probes in the affect recognition logic and the system functional logic, respectively.

b) The implementation of the *functional logic* is open to the developers without restriction, but developers are required to provide the interfaces for extracting information (probes) and for changing the system (effectors).

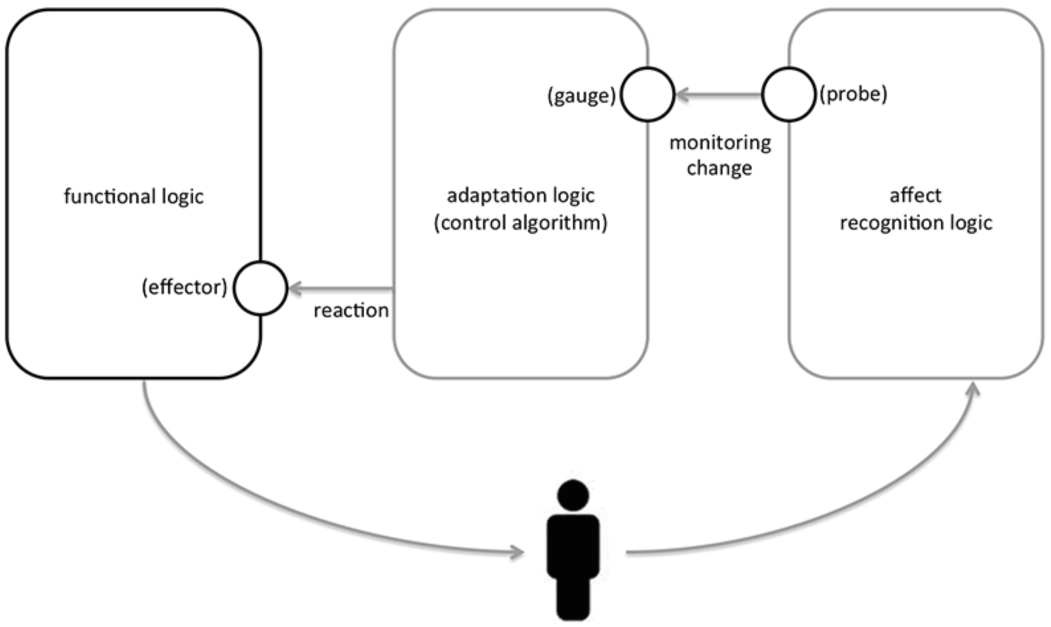


Figure 3.2. Closed-loop architectural model depicting adaptation driven by affective states inferred from a human user.

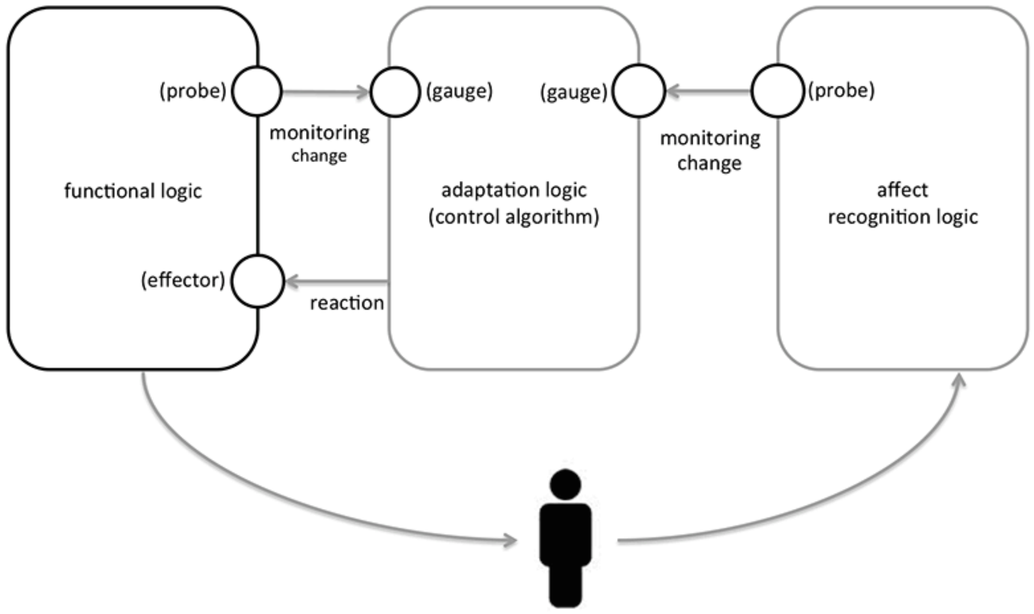


Figure 3.3. Closed-loop architectural model depicting adaptation driven by affective states inferred from a human user and the system context.

3.3. Components

The logics are composed of components that separate concerns with respect to their described functionalities:

a) Components for the functional logic are not described since, as stated before, the functional logic is a wildcard, and can be any new or existing system.

b) The components for the adaptation logic originate in the phases documented in Cheng, Garlan, & Schmerl (2005) for engineering self-adaptation (monitoring, modeling, and controlling), as well as in the categories proposed in Ramirez & Cheng (2010) for dynamically adaptive systems (monitoring, decision-making, and reconfiguration infrastructure).

c) The components for the affect recognition logic and the philosophy behind interface adaptation and affect recognition are based on the affective system patterns presented in previous work (Gonzalez-Sanchez, Chavez-Echeagaray, Atkinson, & Burleson, 2012), but they are updated to take into account the more recent simplification of software infrastructure and process organization. The affect recognition logic has its beginnings in the work of Burleson et al. (2004) and Hussein & Calvo (2009), but it is further improved by the introduction of distributed computing with an agent-based methodology, as well as new and updated sensing devices.

Core components provide the core features for affect recognition (sensing, integration, and synapse). Optional components in the affect recognition logic (perception) and the adaptation logic (reaction engine, introspection, and behavior management) provide optional and customizable features for the products. Figure 3.4 shows the components and their respective logics.

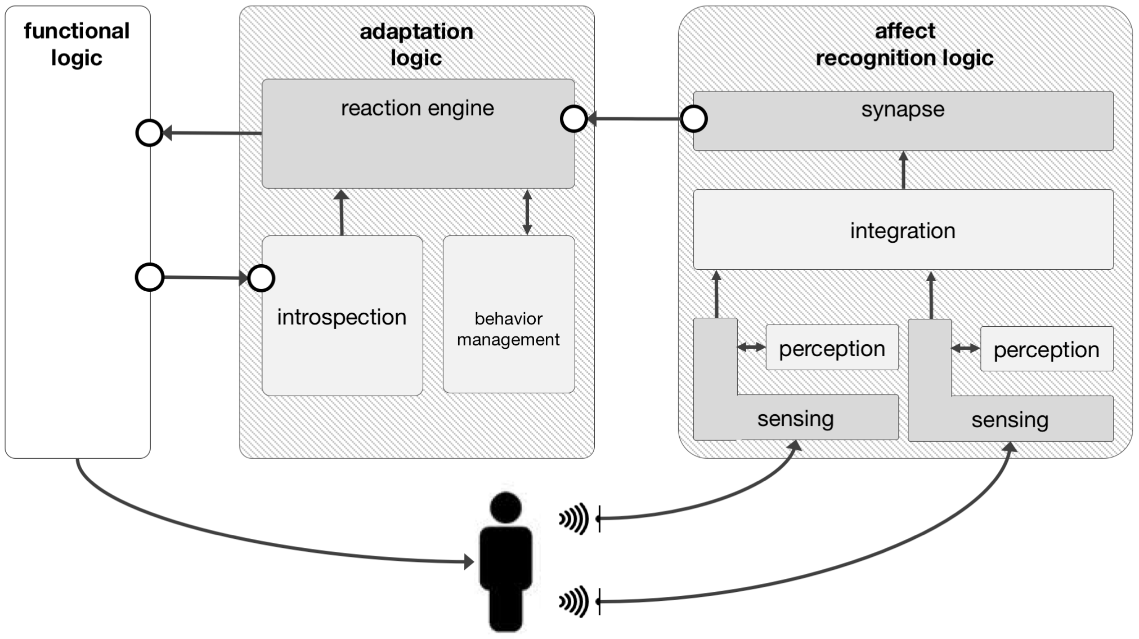


Figure 3.4. Architecture showing components that constitute the affect recognition logic and the adaptation logic – two devices are depicted.

The components in the affect recognition logic resemble patterns from the pattern language for affective adaptive systems, described in section 2.5: sensing, perception, integration, and synapse patterns. Sensing is a functionality that does not require changes after implementation unless the hardware or its communication protocol is changed; however, perception (inference algorithm) changes are commonly made to improve or refine the perception process. As a result, it is advisable to maintain the independence of these functionalities. While not all sensing processes require a perception process, all perception processes are dependent upon a sensing process. For instance, brainwave, pressure, and posture readings need inference algorithms to interpret them, from simple conditionals to more complex approaches using machine learning. Integration makes multimodal affect recognition possible. And, synapse encapsulates the communication responsibilities of the affect logic with other logics.

The components in the adaptation logic are also inspired by patterns from the pattern language for affective adaptive systems, described in section 2.5: reaction engine, introspection, and behavior management. The reaction engine chooses and executes adaptation rules according to the inputs from the synapse component in the affect recognition logic and the introspection component. An action or a sequence of actions is triggered using the effector interface that connects the reaction engine to the target system. When rules are complex, the reaction engine can delegate the responsibility of handling the adaptation rules to a behavior management. Introspection gathers information regarding the system's context (system status, responses, and failures) and UI events from the target system via the probes provided for that purpose.

3.4. The Affect Recognition Logic as an Agent Federation

Considering that affect recognition deals with a diversity of hardware devices and inference algorithms which may draw heavily on computing resources because of their complexity, it makes sense to allow pairs comprising sensing devices and their corresponding inference algorithms to run on different computers if necessary.

A distributed computer model that provides such capabilities is an agent federation, in which units, called agents, autonomously fulfill their responsibilities while they are organized by and report to an agent that acts as a central control unit. In an agent federation organization, the components of the affect recognition logic are visualized in Figure 3.5 and are accommodated as follows:

a) the sensing-perception pairs are encapsulated in the *agents*; sensing and perception have a symbiotic relationship which makes it logical to locate them in a common unit;

b) the *integration* and *synapse* components are located in the central control unit.

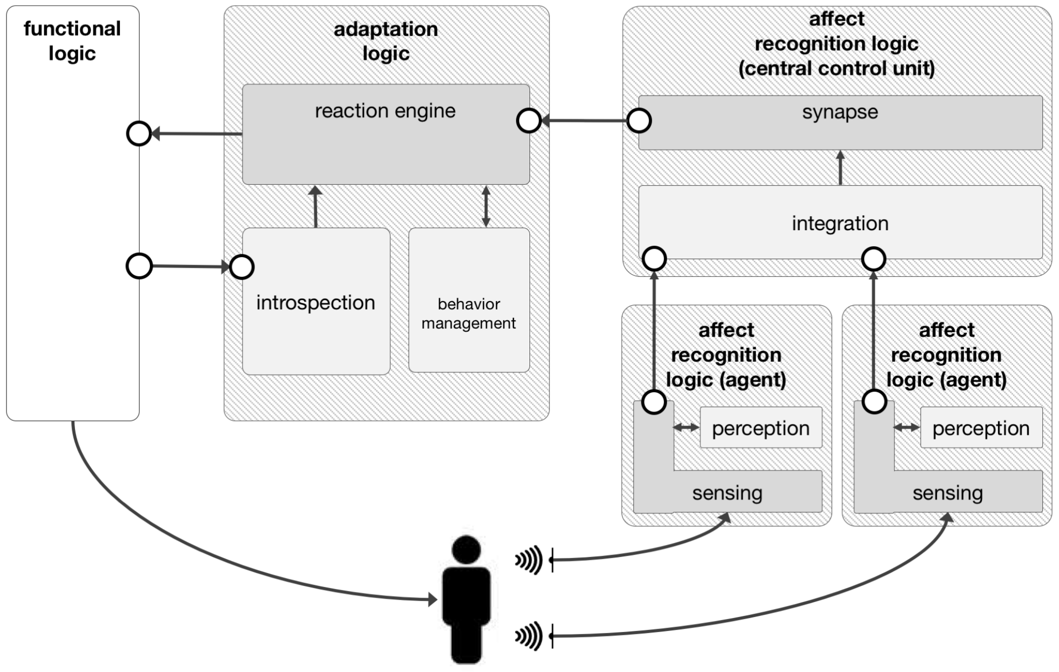


Figure 3.5. Architecture showing the sensing-perception pairs encapsulated as agents. Agents report to a central control unit which encapsulates the integration and synapse components – two agents are depicted.

Another iteration of the model, adding internal components to sensing and integration components, makes explicit: (1) how communication occurs in the federation and (2) how synchronization and fusion are achieved. The complete architecture is shown in Figure 3.6.

The responsibilities of the sensing component are assigned separately to two internal components as follows:

a) the *device handler* gathers values from a hardware device and shares them with the perception component to convert the data gathered to affect measurements.

b) the *publisher* connects with the integration component, specifically with a subscriber component, in the central control unit to report its inferences.

The responsibilities of the integration component are assigned separately to three internal components as follows:

a) The *subscribers* act as counterpart components for the publishers, creating publisher-subscriber pairs, in which the first element (publisher) provides data to the second (subscriber).

b) The *memory* acts as a data repository in which the *subscriber* components deposit the data as it arrives from *publishers*.

c) The *fuser* component takes the data from the *memory* component, synchronizes it, fuses it, and puts the synchronized and fused data back in the memory.

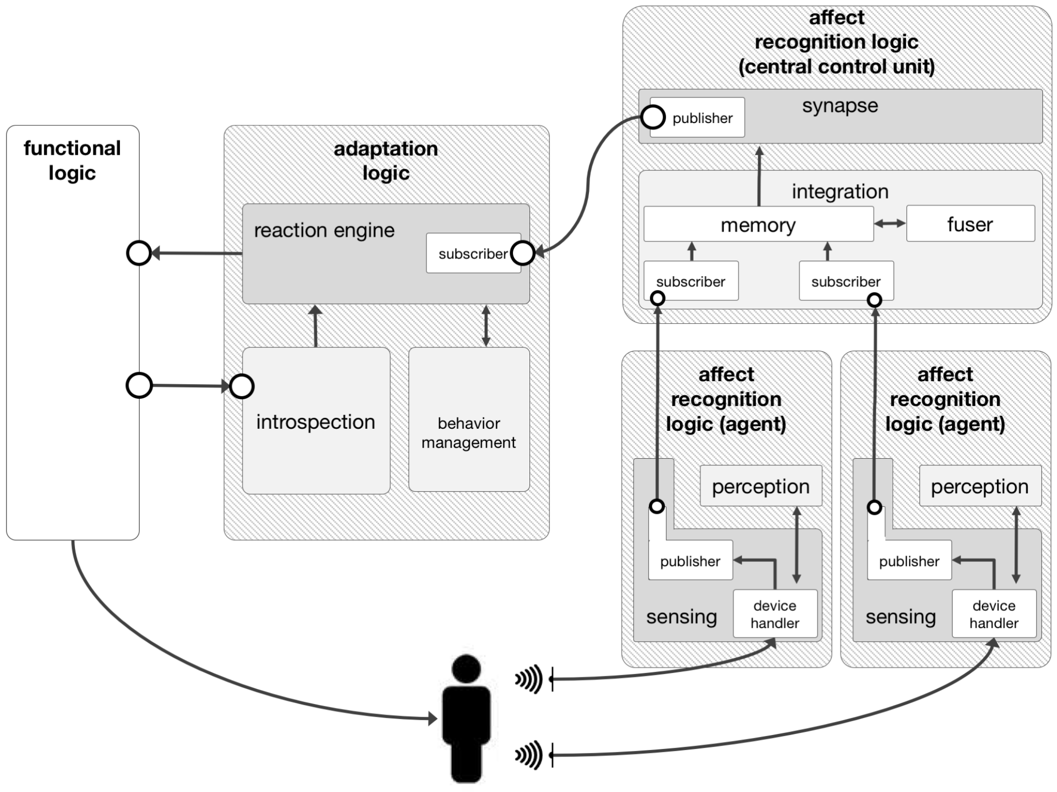


Figure 3.6. Architecture showing the full set of components, including internal components of sensing, integration, and reaction engine components – two agents are depicted.

Finally, the connection between adaptation logic and affect recognition logic also uses the publisher and subscriber components:

a) the *synapse* communicates the resulting value externally using a publisher component; and
b) the reaction engine gathers affect measurements from the affect recognition logic using a subscriber component.

3.5. Scenarios

The architecture covers a range of systems that can be classified using a 2D grid. The first axis considers affect recognition and ranges from lack of affect recognition to monomodal affect recognition capabilities to multimodal affect recognition capabilities. The second axis considers adaptive capability and ranges from no adaptation to simple conditional adaptation to elaborate adaptation using complex algorithms. Nine regions are possible given the possible intersections of the two axes. Within the range of these nine regions, three scenarios are considered, all of which include affect recognition. These are:

a) scenario A includes systems using monomodal affect recognition for affect awareness, but with no self-adaptation capabilities;
b) scenario B includes systems using multimodal affect recognition for affect awareness, but with no self-adaptation capabilities;
c) scenario C includes systems using affect recognition, both monomodal and multimodal, and self-adaptation, both simple and elaborate.

The following three subsections describes these scenarios. The deployment structure of the systems considered in each category is depicted using deployment diagrams where

3D boxes represent hardware hosts linked by network connections (represented by cloud icons). The 3D boxes represent hardware, but not necessarily distinct computers, as the system can run on one computer. This distribution can help to avoid, if needed, the challenge of computing requirements often encountered in affect recognition (sensing devices and inference algorithms). The distribution can help keep these elements from interfering with the adaptation and functional logics. Notice that:

a) The two parts that form the affect recognition logic (central control unit and agents) can run on the same or different computers.
b) Individual agents can run on the same or different computers.
c) The functional logic and adaptation logic exist in the same application and run on the same computer. Therefore, communication between these logics consist of local object-to-object communication or method invocation. The functional logic and adaptation logic can run on different computers, but this setup is not considered in this dissertation; however, there are no limitations to advancing the implementation of the effector and probe-gauge connectors and implementing them as network connections, as is the case in joining the affect recognition logic (central control unit) and the adaptation logic.

3.5.1. Monomodal Affect Recognition for Affect Awareness

The simplest scenario groups systems that have no adaptive capabilities and monomodal affect recognition capabilities. It executes an agent and makes the target system read the value or values reported by that agent directly. Process guidelines are provided to document the development and the network protocol. Figure 3.7 shows a hypothetical system with this configuration. The infrastructure described in the following chapter provides agents for six existing devices and algorithms. A second scenario in this category takes into consideration a hardware device not already included in the infrastructure and therefore requiring the creation of a new agent. The provided infrastructure can be extended to create new agents for new devices. This extension procedure is defined in process guidelines.

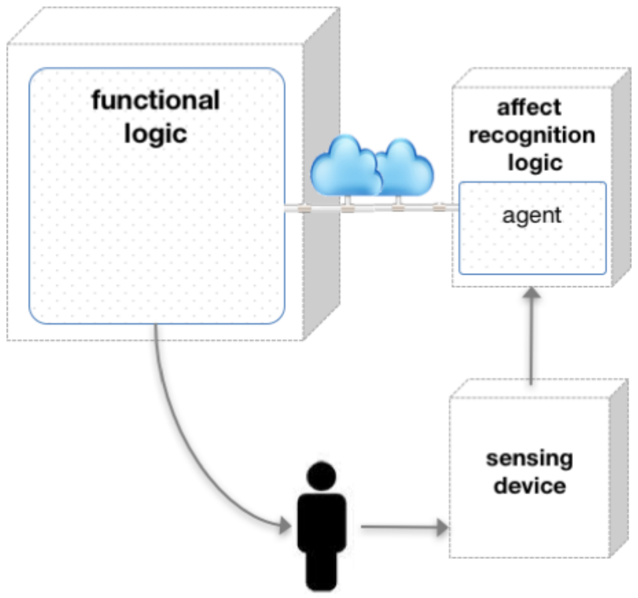


Figure 3.7. Deployment structure for affect aware systems with monomodal affect recognition.

3.5.2. Multimodal Affect Recognition for Affect Awareness

This scenario groups systems that create a multimodal affect-aware target system, in which a set of one or more agents and a central control unit agent are used. The target system reads the fused values reported by the central control unit agent using a network connection. Figure 3.8 shows a hypothetical system with this configuration. The infrastructure described in the following chapter includes agents for six existing devices and algorithms. A second scenario in this category includes a hardware device not already included in the infrastructure or one requiring a different approach for fusing data in the central unit. This scenario requires the creation of a new agent and the modification of the central unit. As mentioned previously, following the architecture, the provided infrastructure can be extended to create new agents for new devices. This extension procedure is defined in the process guidelines.

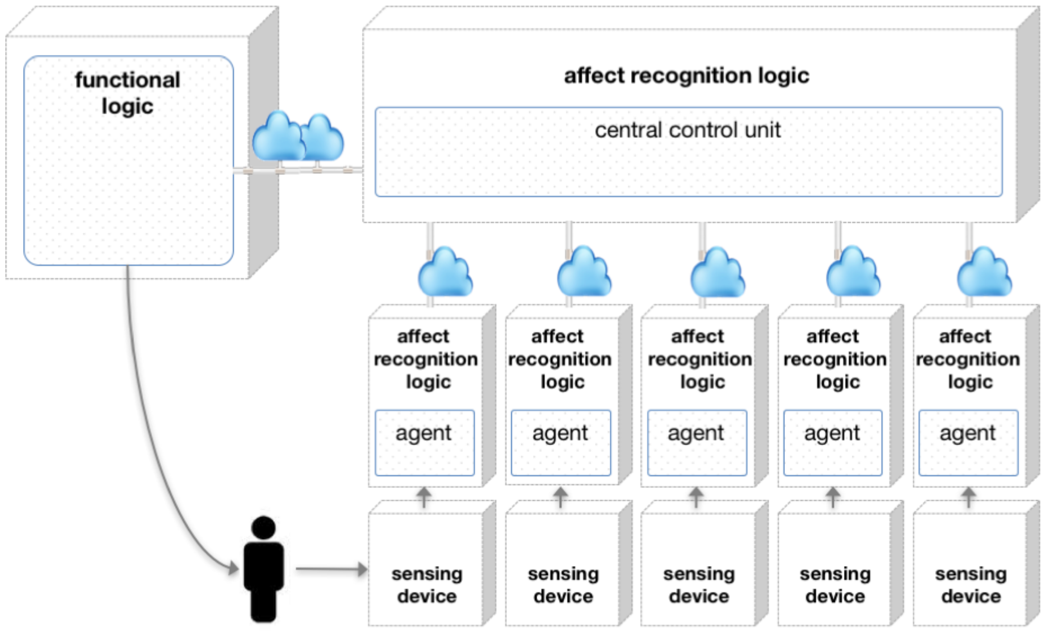


Figure 3.8. Deployment structure for affect aware systems with multimodal affect recognition.

3.5.3. Multimodal Affect Recognition and Adaptation

This scenario groups systems that create a multimodal affect-driven adaptive target system, for which the same procedure is used as for multimodal affect recognition, employing assets from the infrastructure to drive the coding of the adaptation logic and its connection with the target system's functional logic. The complexity of the adaptation rules impact the complexity of the adaptation logic implemented, but the deployment infrastructure is the same in both cases. Figure 3.9 depicts this scenario.

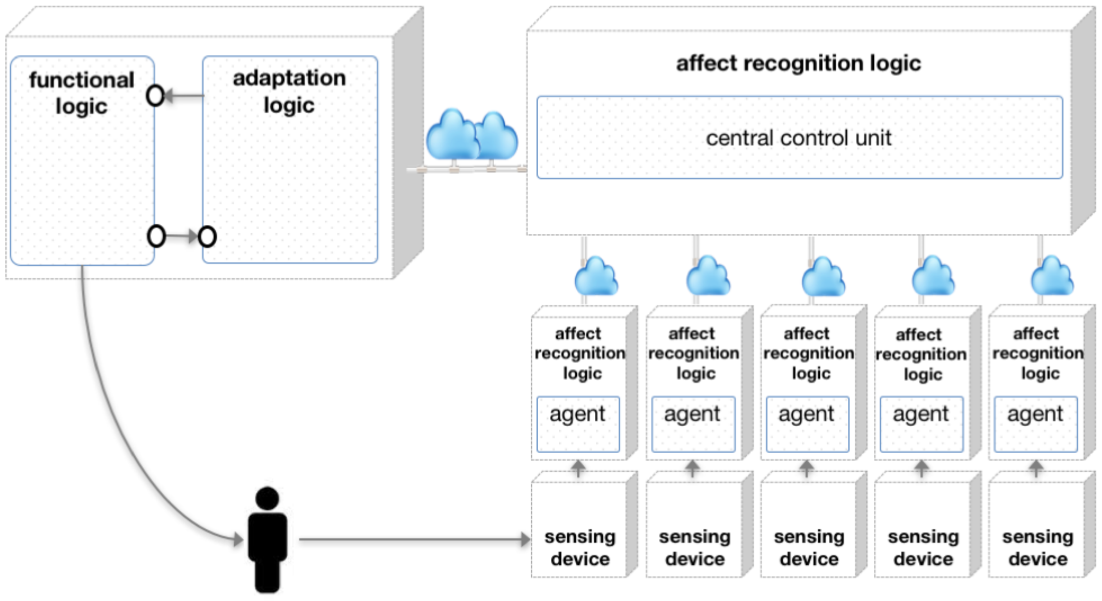


Figure 3.9. Deployment structure for systems with multimodal affect recognition and adaptation capabilities.

3.6. Summary

In this chapter, the tasks and steps in the affect-driven self-adaptation domain were discussed. These drive the design of the ADASPL architecture. The ADASPL architecture supports generality and feasibility by providing a basis for understanding and analyzing the behaviors of affect-driven self-adaptive systems and supporting early design and implementation decisions. The ADASPL architecture defines and connects three logics in a closed-loop relationship. The functional logic encapsulates the complexity of the target system and connects it to the adaptation logic with gauge-and-probe pairs and effectors to the adaptation logic. The adaptation logic consists of three abstract components and is connected to the affect recognition logic by a gauge-and-probe pair. The affect recognition logic consists of four concrete components, and it is modeled as a distributed model using an agent-based approach, due to its complexity. Table 3.1 presents a summary of the components of the architecture. The first column shows the name of the component. The second column states whether the component is a core component or an optional component. The third column states whether the component is a concrete component or an abstract component. The ADAS framework, as described in the next chapter, provides a set of implementations for concrete components but only provides development templates for abstract components; therefore, abstract components are required to be extended for each particular product. The fourth column identifies the components that are required and in which number for each of the three scenarios (A, B, C) described above.

Table 3.1. Components of the Architecture

	Core or optional	Concrete or abstract	Number of possible elements per scenario category		
			A	B	C
Sensing	core	concrete	1	1 - *n*	1 - *n*
Perception	optional	concrete	0 - 1	0 - *n*	0 - *n*
Integration	core	concrete	0	1	1
Synapse	core	concrete	0	1	1
Reaction Engine	core	abstract	0	0	1
Introspection	optional	abstract	0	0	0 - 1
Behavior Management	optional	abstract	0	0	0 - 1

Chapter 4 describes the design and Chapter 5 describes the implementation of six concrete sensing components (and, when required, their optional perception components), an integration component (applying the PAD emotional model), and a synapse component (applying TCP/IP protocol for communication). Further, all components are extensible and Chapter 6 presents guidelines for developers to extend the ADAS framework by creating new implementations of concrete components and completing the implementation of the provided abstract components.

CHAPTER 4
INFRASTRUCTURE DESIGN

The second element of the ADASPL technical solution is an infrastructure that supports generality and cost-effectiveness by providing a well-understood set of reusable and extensible assets. I created such an infrastructure and organized it as a framework named ADAS, for Affect-Driven Adaptive Systems. The ADAS framework supports the deployment of systems with structures described in the scenarios of Chapter 3 and maintains the principle, inherent in the architecture, of keeping the affect recognition and self-adaptation processes independent from one another.

The ADAS framework is founded on previous work that addresses the implementation of the affect recognition infrastructure (Gonzalez-Sanchez, Chavez-Echeagaray, Atkinson, & Burleson 2011a); this previous work is expanded in the ADAS framework to take into consideration advances in affect recognition devices, emotional state model inclusion (PAD), and implementation enhancements that simplify the assets. Also, the ADAS framework is founded on previous work that identifies, in affective systems, phases and their internal steps (Gonzalez-Sanchez, Chavez-Echeagaray, Atkinson, & Burleson. 2011b); these allow affect recognition to be seen as an automated process and facilitate an understanding of how to use ADAS assets to accomplish objectives in multiple scenarios.

The engineering of the ADAS framework drew deeply on the use of software design patterns. It follows a component-based methodology and the object-oriented paradigm (Crnkovic 2001; Johnson 1992; Schmidt & Buschmann 2003; Sommerville 2002; Kruchten 2004; Clements, Kazman, & Klein 2001). It was an iterative process that led to the creation of packages – namespaces that organize a set of related classes and interfaces. The following packages are included in the ADAS framework:

a) a package for each of the six components described in the architecture,

b) a package encapsulating common tasks (named `toolkit`), and

c) a package encapsulating useful GUI elements (named `gui).`

Using these packages tools, in the form of pre-built ready-to-use applications, are assembled to support the affect recognition processes. These tool applications help developers to get started and all have a GUI to facilitate easy adoption. The following tool applications are included in the ADAS framework:

a) a set of agents for the previously described devices,

b) a central control unit with a PAD emotional state model as a fusion approach, and

c) a dashboard to facilitate launching them, when all are used on a single computer.

The ADAS framework is implemented on the Java platform that provides the application's programming interface, including infrastructure for handling multithreading and networking. It uses external native libraries to interface with commercial off-the-shelf software (including Emotiv SDK and Tobii SDK) and to deal with hardware devices connected through serial ports or Bluetooth (some of them developed in-house). A 32-bit Java Standard Edition Platform (JSE) was employed, using Java Software Developer Kit (JDK) version 8. A 32-bit platform was required since some of the external libraries were not available in 64-bit versions. The Java platform was chosen for its portability; however, this portability was limited by the required use of external native libraries that are only available for Windows environments. This limitation impacts on the pre-built ready-to-use agents, which are restricted to running in Windows environments; the central control unit, the adaptation logic, and the functional logic are all open to other platforms.

The ADAS framework structure is depicted in Figure 4.1 and summarized below:

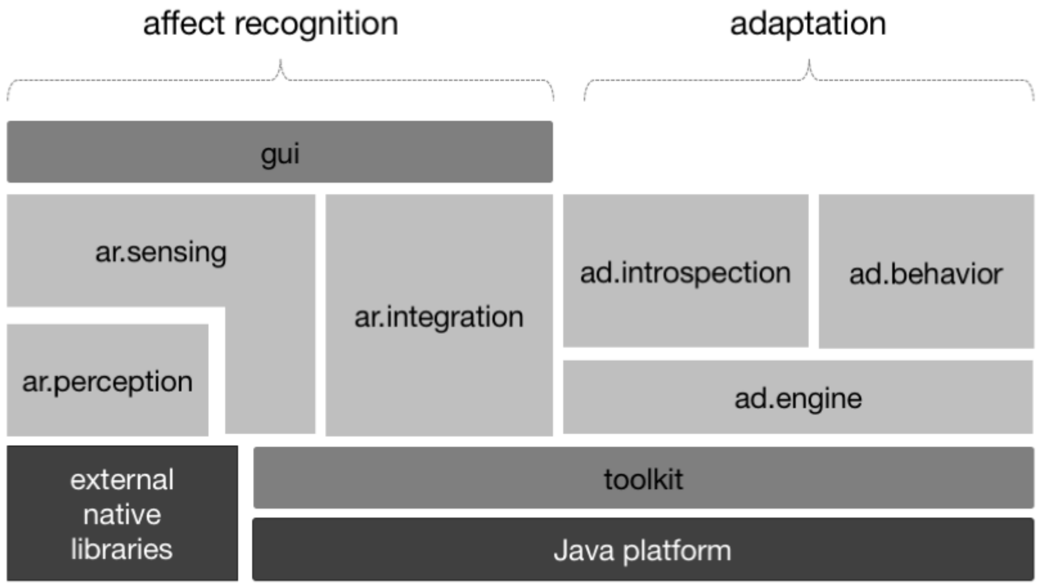


Figure 4.1. Structure of the ADAS framework.

a) Package `ar.sensing` contains classes and interfaces for creating new agents and for instantiating agents that handle the following devices: Emotiv EPOC headset, Tobii T60XL eye-tracking system, MindReader facial emotion recognition, skin conductance device, pressure sensing device, and posture sensing device. It encapsulates the access to the external native libraries for those devices.

b) Package `ar.perception` contains classes and interfaces to define or incorporate perception algorithms, some of which require wrapper classes to access external native libraries.

c) Package `ar.integration` contains classes and interfaces for fusing values using a blackboard-based approach. A fuser based on the PAD emotional state model is provided.

d) Package `ad.instrospection` contains interfaces that support the implementation and integration of introspection capabilities, including the infrastructure to bridge the adaptation and functional logics.

e) Package `ad.engine` contains interfaces that support the implementation and integration of adaptive reaction capabilities.

f) Package `ad.behavior` contains interfaces that support the implementation and integration of complex adaptation rules.

g) Package `toolkit` contains classes and interfaces that implement common tasks for network communication and data abstractions. The capabilities of the synapse component are embedded in this package.

h) Package `gui` provides customized widgets for data visualization, including plotting, charts, and customized consoles.

In the following sections the internal composition of the packages and their internal design is detailed; packages related to affect recognition and adaptation are described separately. Descriptions of the `gui` and `toolkit` package contents are presented in conjunction with the packages that use them in the affect recognition and adaptation infrastructure. Finally, the tool applications are presented.

4.1. Affect Recognition Infrastructure

The affect recognition infrastructure facilitates the use of existing affect recognition methods by either encapsulating previous implementations or providing interfaces to enable new implementations as needed. It is formed by the `ar.sensing`, `ar.perception`, and `ar.integration` packages, the first two of which constitute the agent and the third the agent central control unit, as shown in Figure 4.2. The `toolkit` and `gui` packages support both the agent and the central control unit. The following two subsections describe the packages, grouping them by their use in each element of the affect recognition infrastructure: agents and central control unit.

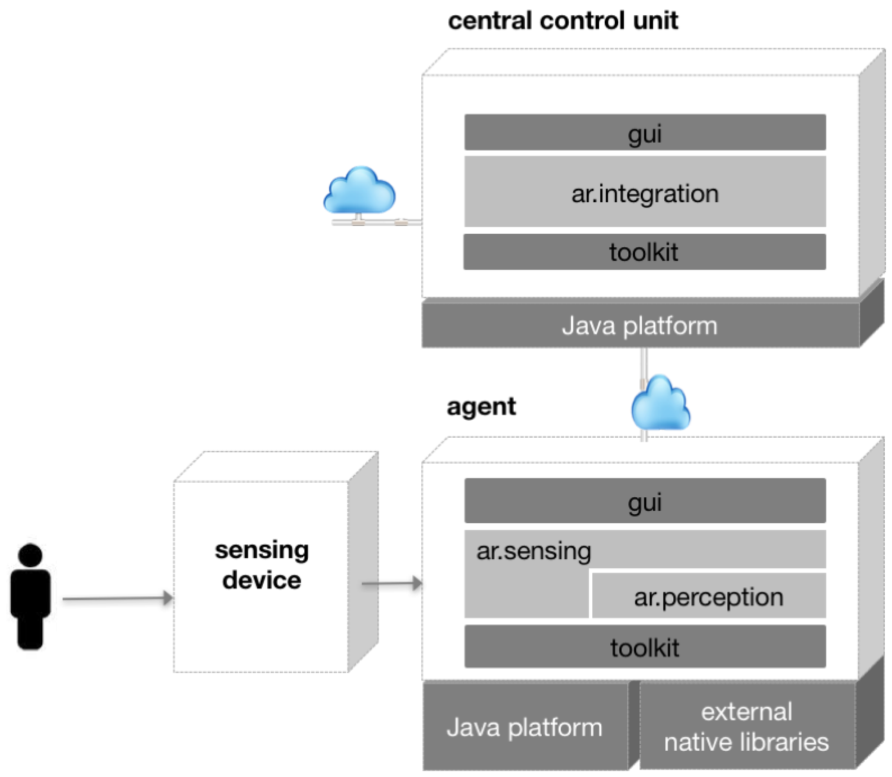


Figure 4.2. Assembly of the affect recognition logic (central control unit and agents) with the packages provided in the ADAS framework.

4.1.1. Agents

Agents collect values from hardware devices and make them available through a network port. The core of each agent is composed of the sensing and perception components; both are considered core components since they are needed for all products of the product line. Two parameters should be configured in an agent:

a) the device or source of the data, and

b) the computer address to which the agent publishes its inferences.

Creating, modifying, or extending agents requires an understanding of their internal structure, i.e., the classes and interconnections inside their packages. The internal structure of the components is based on software design patterns, and therefore can be understood as a combination of them, particularly,

a) a *Facade* pattern for encapsulation;

b) an *Observer* pattern for internal communication, where instances that want to share information declare themselves as Observable and instances interested in the information declare themselves as Observers;

c) a *Delegate* pattern to separate sensing and perception processes;

d) a *Publisher-Subscriber* pattern for network communication, where Publishers send information using a network connection and Subscribers receive information using a network connection; and

e) a *Singleton* pattern to ensure that only one instance can be created for a particular class.

Further, agents use multithreading, and therefore they can collect and communicate values at the same time and several agents can run at the same time. The `Runnable` interface from the Java platform is used to implement multithreading instances.

There are two categories of agent, defined by how they connect with the sensing device:

a) Agents that connect with the device using a network port, named AgentNet. The pre-built agents for the Emotiv Composer and headset, Tobii eye tracker, and MindReader belong to this category.

b) Agents that connect with the device using a serial port connection, named AgentComm. The pre-built agents for skin conductance, posture, and pressure sensing devices belong to this category.

A graphic representation of the classes inside each package, for each category, is shown in Figures 4.3 and 4.4, and described below referencing the software design patterns that are used with a *stereotype* label.

The classes and their responsibilities are as follows:

a) `ThreadDevice` (in AgentNet) and `ListenerDevice` (in AgentComm) are execution threads; they implement the `Runnable` interface and gather information from a network port and a serial port, respectively. They share information with the other classes using the *Observer* pattern infrastructure. To declare themselves as a source of information they extend the `Observable` class.

b) `ThreadDevice` and `ListenerDevice` delegate the transformation of the gathered data into affect measurements to a helper object from `DelegatePerception` class. `DelegatePerception` hosts the inference algorithms. They are connected using a *Delegate* pattern infrastructure. If no transformation is required, `DelegatePerception` is null. For instance, when gathering Emotiv affective constructs, no additional transformation is required. Inferred values are stored together with the gathered values, i.e., kept it in the same data object. `DelegatePerception` is an interface, and so is an extension point.

c) `SerialPortManager` is implemented using the *Singleton* pattern to guarantee the existence of only one manager for the serial ports in the computer. The manager, when created, rasterizes all serial ports in the computer, searching for known devices (currently skin conductance, posture, and pressure sensors). Then, it creates a list of connected devices and the port to which they are connected. `ListenerDevice` will consult that list to locate the physical device from which it has been requested to gather data.

d) Both AgentNet and AgentComm have a GUI with a group of plotting panels (class `Plot` instances), contained in a `PlotSet` instance, and a console showing data in CSV format, provided as a `ConsoleObserver` instance. Both `PlotSet` and `ConsoleObserver` extend `JPanel` and get their data by observing a `ThreadDevice` or `ListenerDevice` instance. Thus, an *Observer* pattern guides the interconnection of `PlotSet` and `ConsoleObserver` with a `ThreadDevice` or a `ListenerDevice`, and information that arrives to `ThreadDevice` or `ListenerDevice` can immediately be shown on the GUI.

e) `PublisherThread` is also an Observer of a `ListenerDevice` or a `ThreadDevice` instance. It is responsible for sending the values gathered and inferred by the agent to the central control unit. It observes new data in `ListenerDevice` or `ThreadDevice` and streams it to the Subscribers. `PublisherThread` is half the implementation of a *Publisher-Subscriber* pattern.

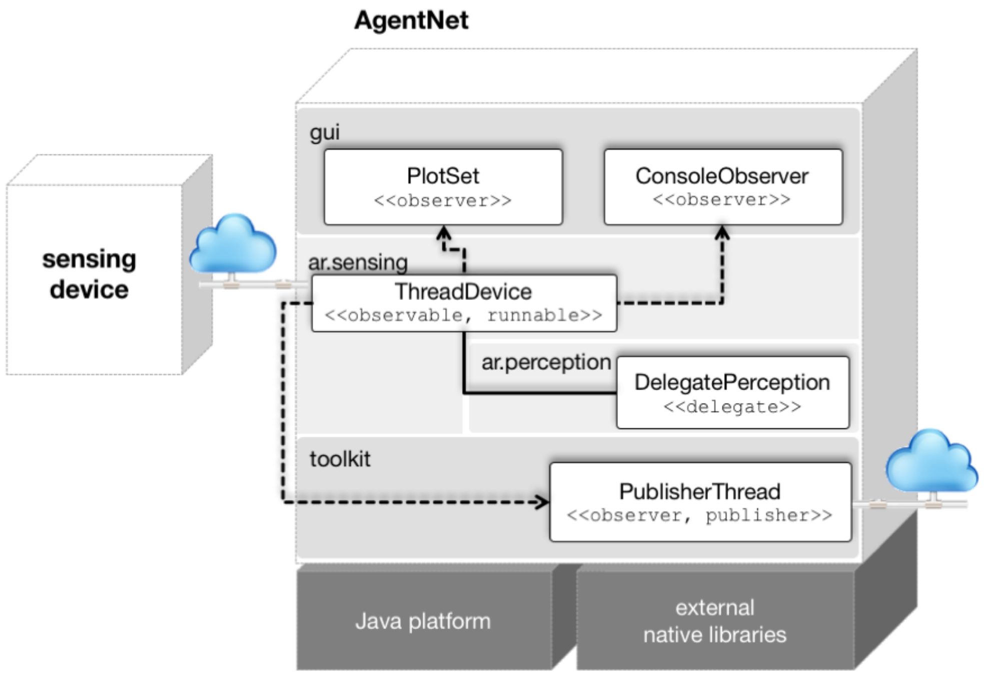


Figure 4.3. AgentNet structure with *Facade*, *Observer*, *Delegate*, and *Publisher-Subscriber* patterns. Notice the use of a `ThreadDevice` instance to connect with the device via a network connection (represented as a cloud).

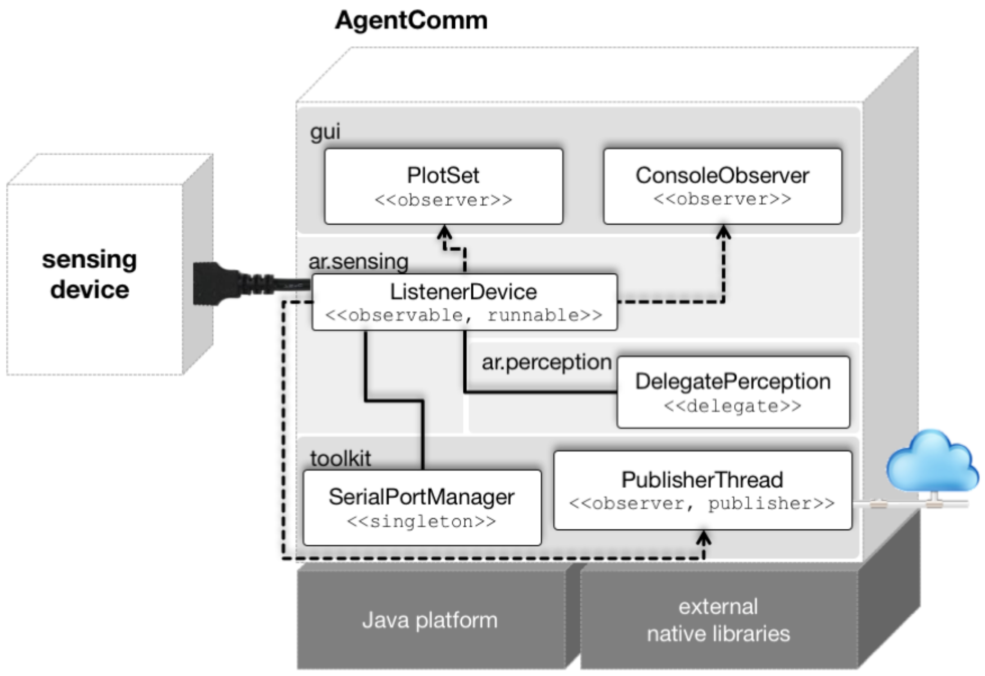


Figure 4.4. AgentComm structure with *Facade*, *Observer*, *Delegate*, *Singleton*, and *Publisher-Subscriber* patterns. Notice the use of a `ListenerDevice` instance to connect with the device via a local port (represented as a cable).

4.1.2. Central Control Unit

The central control unit is an agent that governs the agents that connect with the hardware devices. Its core is composed of the integration and synapse components. The central control unit is optional to products that would like to take advantage of a multimodal approach. It receives data from running agents and creates an output stream with the integrated inferences to a local port.

The central control unit gathers data from running agents, creating connections to the addresses enumerated in Table 4.1 in the column output. The port number to which inferences are published is a parameter that should be defined in configuration. The pre-built central control unit is ready to be used without modification and includes:

a) a GUI showing plots and a 3D view of the cumulative PAD values in a `Canvas` instance;

b) a memory component providing a shared repository for data values; and

c) a fuser component customized to use the PAD emotional state model as a fusion approach and a state machine synchronization approach, where the last value received from a source is considered still present until a new one arrives.

Modifying and extending the central control unit requires an understanding of its internal structure. The internal structure of the components in the central control unit is based on software design patterns, and therefore can be understood as a combination of them, particularly,

a) a *Facade* pattern for encapsulation;

b) a *Blackboard* pattern to provide a shared repository for data values and to coordinate separate, disparate systems that need to work together;

c) a *Delegate* pattern to separate the fusion process from the data collection; and

d) a *Publisher-Subscriber* pattern for network communication, where Publishers send information using a network connection and Subscribers receive information using a network connection.

A graphic representation of the classes inside each package is shown in Figure 4.5 and summarized below. The classes and their responsibilities are as follows:

a) Instances of `SubscriberThread` listen to `PublisherThread` instances created on the agents. The `SubscriberThread` uses an *Observer* pattern implementation to notify the `Blackboard` when a new value arrives.

b) The `Blackboard` instance stores a PAD representation of each received value. It provides a dynamic data structure that stores the most recent value reported from each agent. When a new value is received, the previous one is replaced. The `Blackboard` class belongs to the core of a *Blackboard* pattern implementation.

c) The `FuseDelegate` is an interface used to create the supervisor role, as defined in the *Blackboard* pattern. It is called each time that a new value arrives to the `Blackboard` container and applies the synchronization and fusion algorithm to combine all received values into one multimodal measure. An implementation of the `FuseDelegate` interface is provided that synchronizes PAD vectors following a state machine approach and fuses PAD vectors by adding them together. The resulting value is also put into `Blackboard` as a `PAD` object. The `Blackboard` instance shares its stored values using an *Observer* pattern implementation; it is defined as Observable, while `PlotSetPAD`, `Canvas`, and `PublisherThread` instances are Observers.

d) A `PublisherThread` accesses the fused `PAD` object in `Blackboard`, implementing the `Observer` interface and communicating it externally.

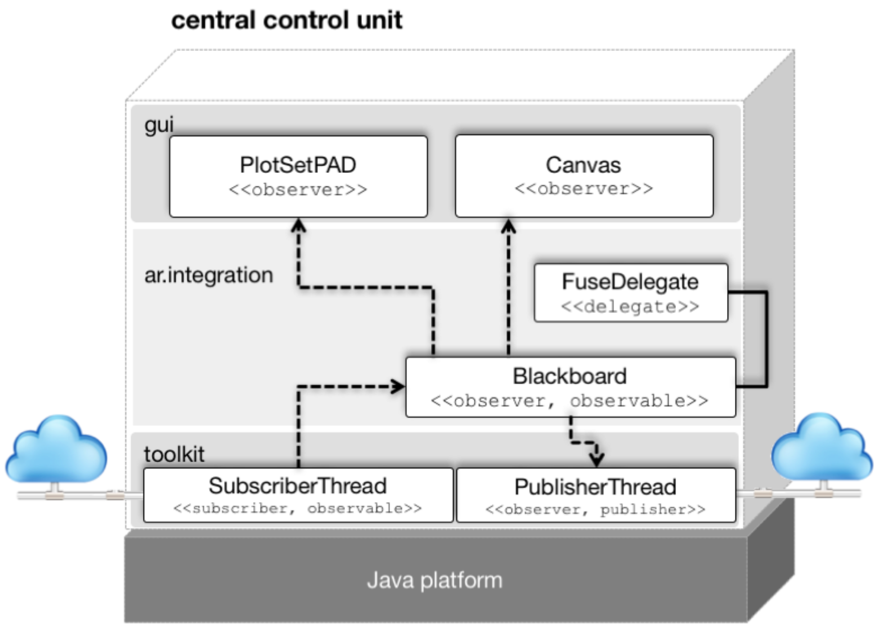


Figure 4.5. Central control unit with *Facade*, *Blackboard*, *Delegate*, and *Publisher-Subscriber* patterns. Cloud icons represent input connections from agents and an output for sharing inferences.

4.2. Adaptation Infrastructure

The adaptation logic components are abstract components; therefore, developers must provide an implementation extending abstract classes and interfaces. Internally, the adaptation logic components consist of two classes to interface with the functional logic (`Actuator` and `Introspector`), and an abstract class and an interface (`Engine` and `Manager`, respectively) that outline the adaptation rule implementation approach and define extension points. Moreover, an instance of `SubscriberThread` class allows the adaptation logic to gather affect measurements from the central control unit (specifically from the synapse component in the affect recognition logic) via a network port.

The adaptation rule implementation approach deals with diverse ranges of complexity: it can be achieved by using a simple set of nested if-else statements or by using sophisticated third-party systems interfaced with the adaptation logic. For instance, there are third-party systems that allow the definition and execution of condition-action rules and third-party systems that use temporal logic to define alternative weighted decision paths.

The internal structure of the components in this logic is based and can be understood as a combination of the following software design patterns: *Observer*, *Delegate*, and *Publisher-Subscriber*. A graphic representation of this internal structure (classes and interfaces inside each package) is shown in Figure 4.6 and summarized below.

a) `SubscriberThread` gets data from emotion recognition using a *Publisher-Subscriber* pattern and shares it with `Engine` using an *Observer* pattern.

b) `Introspector` gets data from the functional logic using probes. Probes pack information in `Event` instances and notify the `Introspector` instance of their creation. The `Event` class encapsulates the information that the functional logic wants to share with the adaptation logic. The `Introspector` instance handles events and then makes results available to the `Engine` instance. To do so, an *Observer* pattern implementation is applied.

c) `Engine` receives affect data from the `Subscriber` instance and receives user-event updates from the `Introspector`. It is an abstract class, i.e., an extension point, whose functionality must be defined by developers. It delegates the support for complex adaptation rules to `Behavior` instance. It calculates an action to be triggered, which is encapsulated in an `Action` instance and posted to the `Actuator`.

d) `Actuator` instance handles actions; it is observed by the functional logic and triggers reactions there.

e) `Manager` is an optional component for systems that require a complex set of adaptation rules. When complex adaptation rules are needed, it may be desirable to separate the adaptation rules from the system and have a specialized support to handle them; thus, modifying the rules does not mean modifying the system itself. External support to handle adaptation rules can be provided by a production rule engine. `Manager` is an interface that developers can extend to implement specialized functionalities. The `Engine` class uses a `Manager` instance set to null by default, i.e., it is not required.

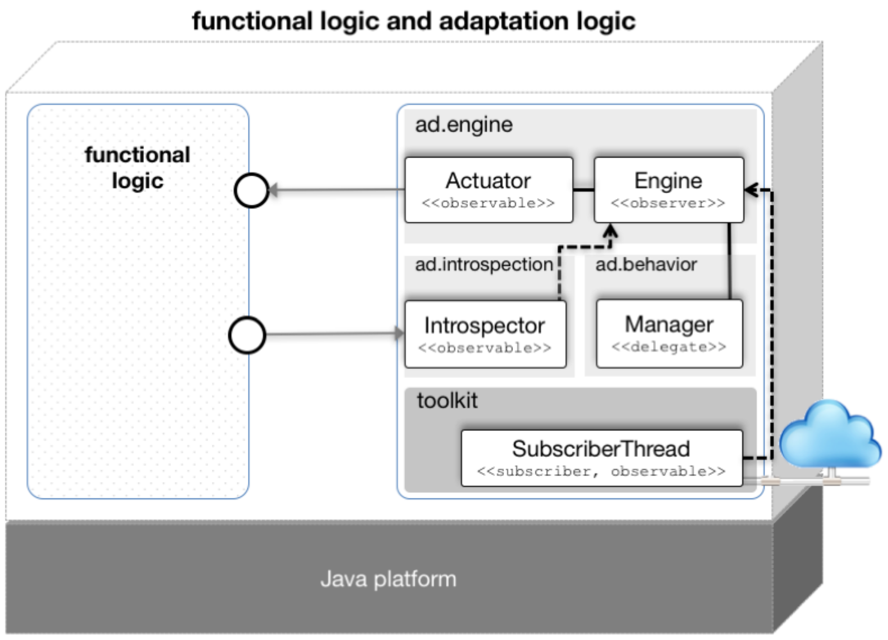


Figure 4.6. Adaptation logic with *Observer*, *Delegate*, and *Publisher-Subscriber* patterns.

4.3. Tool Applications

Using the described infrastructure of components, a set of agents, a central control unit, and a dashboard to test them was created. This served as a starting point for developers and supports the broader adoption of ADAS. The tool applications included are listed below.

4.3.1. Agents

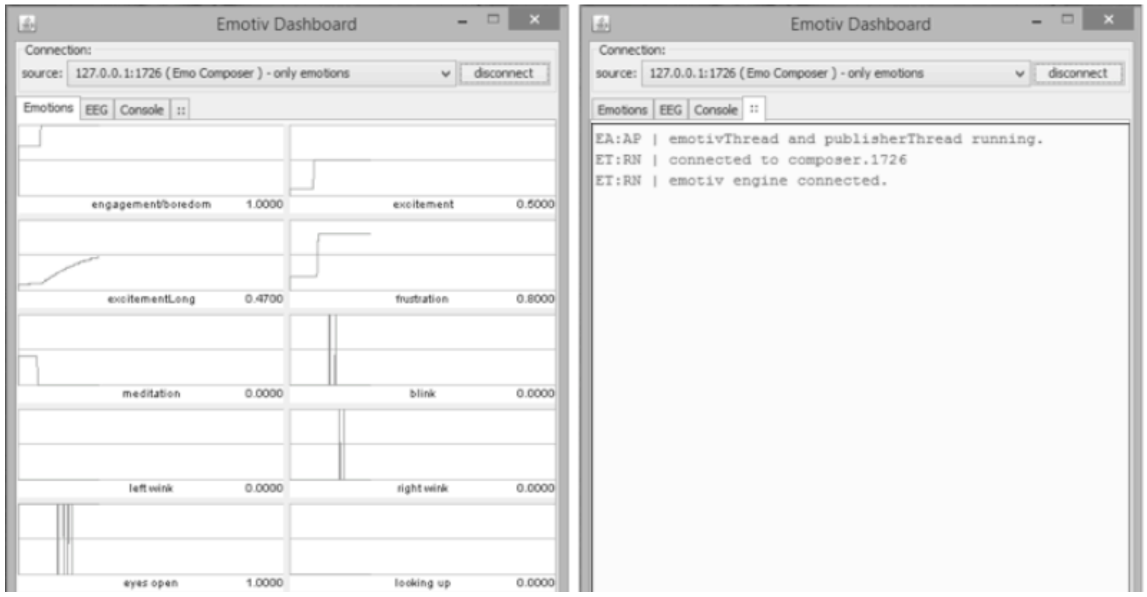
Six fully implemented agents are provided as tool applications; they are ready to be used without modification. They support the devices listed below; details of the devices and their data are described in Gonzalez-Sanchez et al. (2011c) and summarized in section 2.2.

a) An agent application able to gather data from the Emotiv EPOC headset and the Emotiv Composer.
b) An agent application able to gather data from the Tobii T60XL eye tracker.
c) An agent application able to gather data from MindReader.
d) An agent application able to gather data from a skin conductance device.
e) An agent application able to gather data from a pressure-sensing device.
f) An agent application able to gather data from a posture-sensing device.

All agent applications have a similar GUI that allows developers to

a) connect and disconnect the agent and the hardware;
b) show the gathered data in plots;
c) show the gathered data as a CSV string on a console; and
d) show the connection status and system messages on a console labeled as ‘::’.

Figure 4.7 shows the GUI of the agent application for Emotiv. Each agent is intended to run on the computer to which its corresponding hardware device is connected. An exception is the Tobii T60XL system, which streams its information via its own network connection. Table 4.1 shows the default input and output addresses for the pre-built agents.



source: 127.0.0.1:1726 (Emo Composer) - only emotions

```
EA:AP | emotivThread and publisherThread running.
ET:RN | connected to composer.1726
ET:RN | emotiv engine connected.
```

Figure 4.7. GUI of the agent application for the Emotiv EPOC headset and Emotiv Composer. The GUI includes a tabbed panel that, in real-time, displays and plots the emotional constructs and the raw EEG data collected.

Table 4.1. Configuration of the Provided Application Agents

Source	Input	Output
Emotiv Composer	<composer URL>:1726	localhost:7676
Emotiv Headset	<headset URL>:3008	
Tobii	<jtobiistream URL>:11475	localhost:7031
MindReader	<jmrstream URL>:11476	localhost:7032
Skin	auto detection of a port	localhost:8080
Posture	auto detection of a port	localhost:8081
Pressure	auto detection of a port	localhost:8082

4.3.2. Central Control Unit

The central control unit agent gathers information from running agents and implements a fusion method based on the PAD emotional state model, which maps constructs to a coordinate vector and then adds all the coordinate vectors together, as described in Chapter 2. The PAD emotional state model is the affective formalism implemented in ADAS. It allows the mapping of multimodal affect reports as PAD vectors, adding the set of vectors to produce a single affect report. The selection of this 3D model is based on the need to standardize features from diverse sensing devices in real-time systems (Gilroy, Cavazza, & Benayoun 2009). Additionally, the central control unit can be run disconnected from agents to generate and report random PAD values that can be used for testing purposes. Figure 4.8 shows the central control unit agent GUI, which includes three tabbed panels:

a) the first displays PAD values in a 3D space enclosed in a cube;
b) the second shows a chart plotting pleasure, arousal, and dominance values; and
c) the third tab, labeled '::', has a console that shows messages relating to the system status, including errors or failures. When running in simulator mode or with at least one agent connected, the central control unit streams PAD inferences to the following ports in the host computer by default: 7474 using the CSV protocol and 7575 using the serialized Java object format.

4.3.3. Dashboard

The ADAS dashboard provides a common GUI for the agents and the central control unit. It facilitates the execution of agents and the central control unit in the same computer.

Instead of running each of them as an independent application, i.e., having multiple windows on the screen, the dashboard allows one application to

a) execute agents for diverse devices and organize their GUIs in vertical tabs in a single window;
b) execute the central control unit agent; and
c) monitor the connection status and system messages on a common console.

Beside the label in each tab is an on/off symbol that updates to show whether or not the corresponding agent is running, changing color from green (to indicate on) to gray (to indicate off). Figure 4.9 shows the dashboard GUI. The image shows the agent for the Emotiv Composer running. It is gathering emotional constructs only, so the 14 raw data channels appear empty. Inside the tab for each agent, there are two additional tabs: one to visualize plots and another to visualize the data as CSV strings.

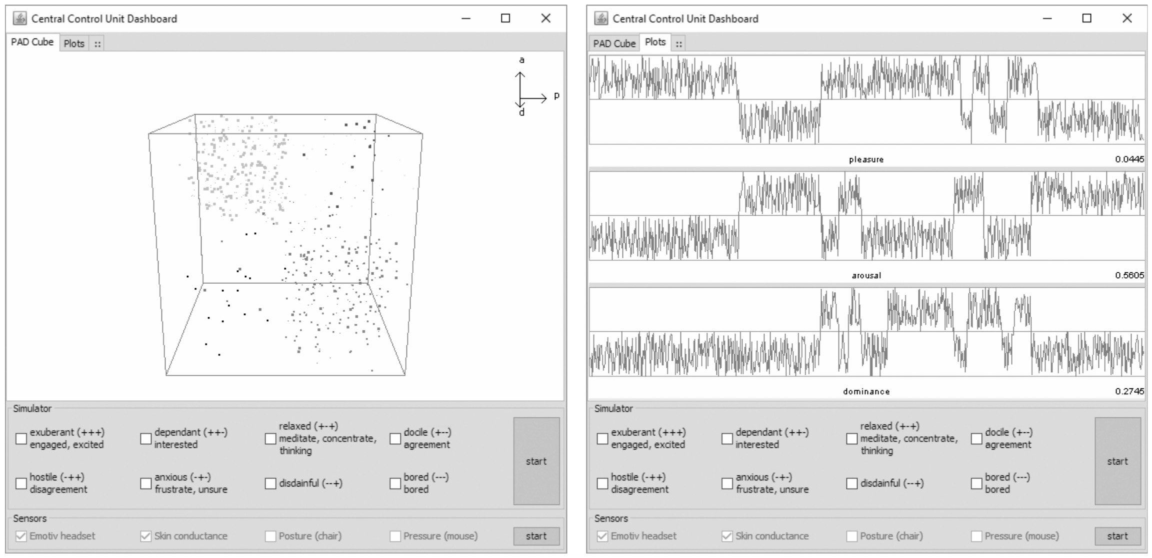
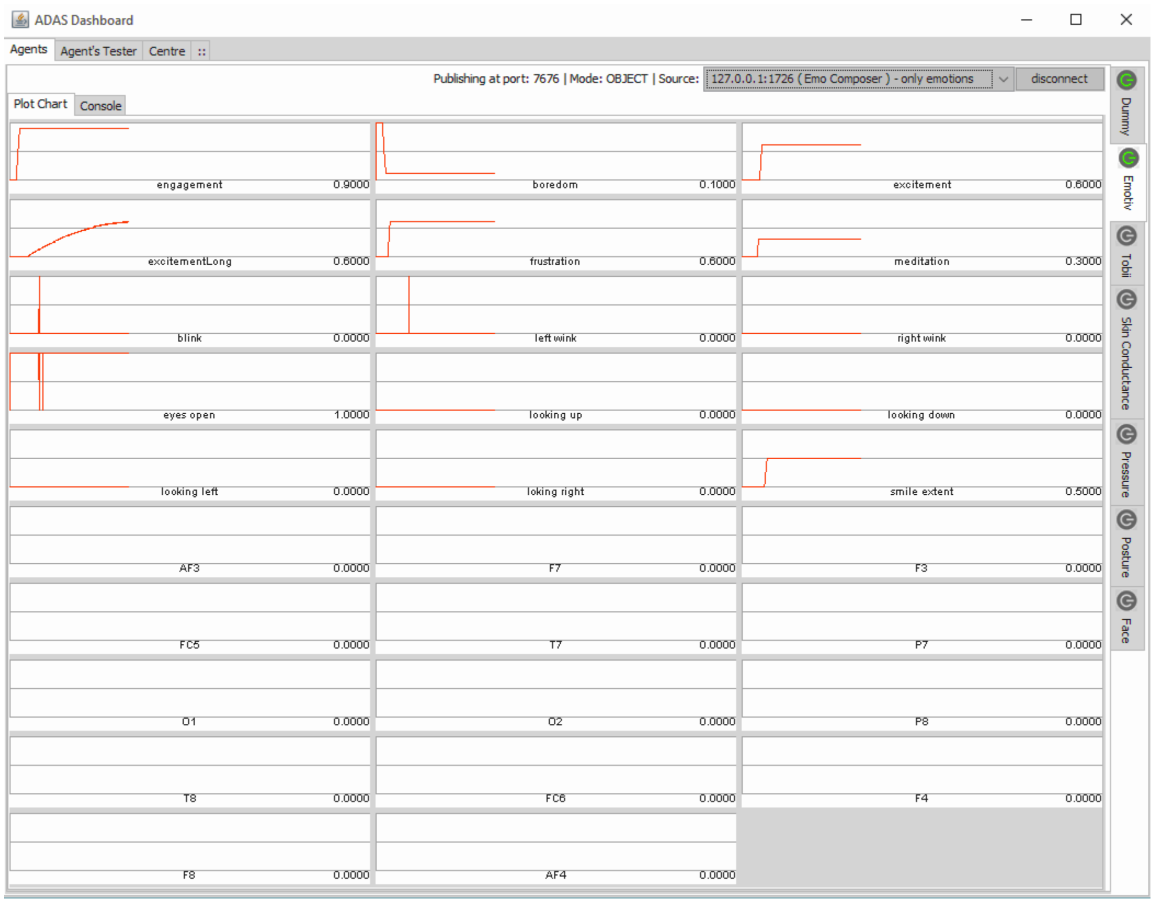


Figure 4.8. GUI of the central control unit dashboard application showing two of the three panels: the first panel displays PAD values in a 3D space; the second panel displays PAD values in plots; and the third panel (not shown in the figure) is used to display messages from the system console.



Publishing at port: 7676 | Mode: OBJECT | Source: 127.0.0.1:1726 (Emo Composer) - only emotions

Figure 4.9. GUI of the ADAS dashboard application.

4.4. Summary

In this chapter, I introduced the ADAS software framework which supports generality and cost-effectiveness by providing a well-understood set of assets for affect recognition and closed-loop adaptation. In particular, I detailed the set of components that constitutes the infrastructure, presenting the key classes of each component as well as their interconnections, founded on software design patterns. I also described how the designed infrastructure allows the construction of systems that have a deployment structure that corresponds to one of the three scenarios described in the previous chapter. Affect recognition can be added to systems either by using the provided applications for already integrated hardware, or by extending the infrastructure to add new sensing hardware and new inference and integration algorithms. The common ground for adaptation capabilities is established so that the particular necessities of the target system can be built upon this. The next chapter discusses in detail the implementation of the design described here.

CHAPTER 5.
INFRASTRUCTURE IMPLEMENTATION

The previous chapter points out how software design patterns underlie the design of packages and also describes their key classes, as well as the interfaces and their connections. Extending or modifying ADAS requires an understanding of all of these classes and interfaces. This chapter introduces the full hierarchies of the classes and interfaces that constitute the ADAS cohesive software design, providing:

a) a complete object-oriented design, shown in UML class diagrams;

b) a glance at class and interface content with snippets of code; and

c) a description of the data encapsulation assets.

5.1. Package ar.sensing

The hierarchy of classes for the sensing package is depicted in Figure 5.1. Red circles highlight classes already mentioned in the previous chapter.

5.1.1. Interface Agent

The `Agent` interface defines a contract establishing what an agent implementation must be. It contains the signatures for methods `start`, `stop`, `shutdown`, and `getObservableSource`. The purpose of `start`, `stop`, and `shutdown` is to perform the named action. The method `getObservableSource` provides access to an `ObservableSource` instance (a `ThreadDevice` or a `ListenerDevice` instance). The `Agent` interface is used to implement `AgentNet` and `AgentComm` classes.

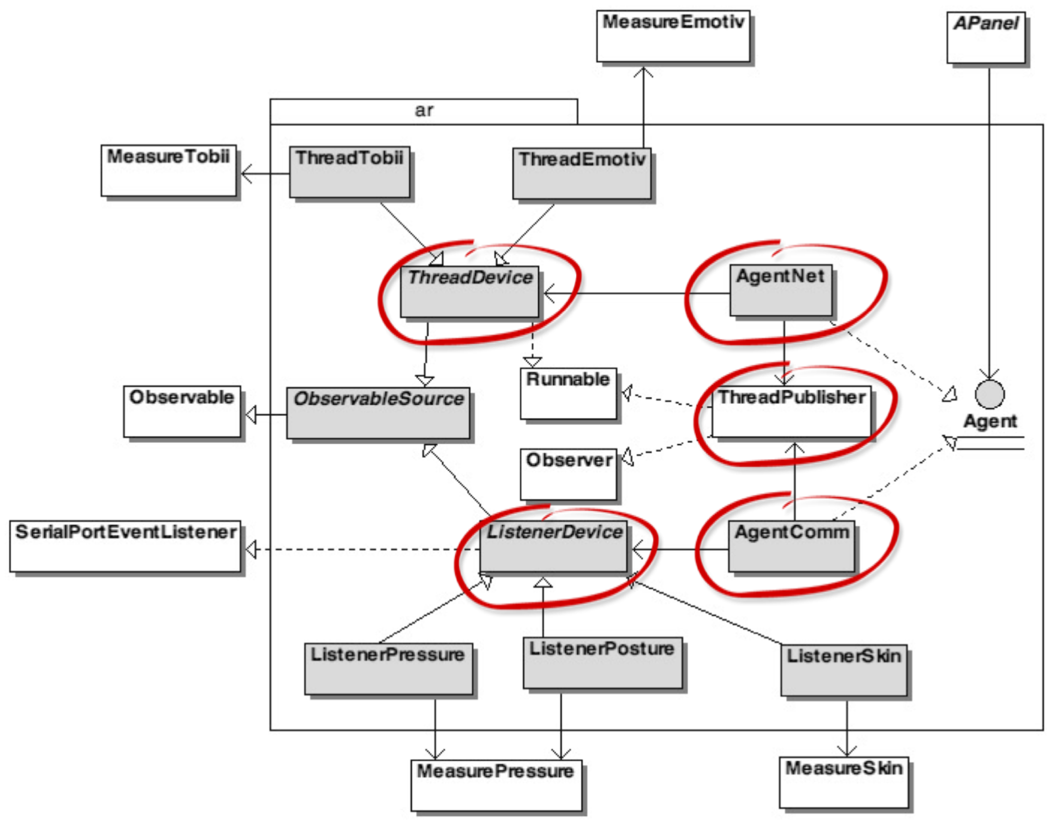


Figure 5.1. UML diagram of the package `ar.sensing`. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

```
package adas.ar.sensing;

public interface Agent {
  public void shutdown();
  public boolean start();
  public void stop();
  public ObservableSource getObservableSource();
}
```

5.1.2. Class ObservableSource

The abstract class `ObservableSource` forces the classes that implement it to have a getter method `getObject` that gives access to a `Measure` instance (the value being observed). The `ObservableSource` class extends from `Observable` to define itself as a subject to be observed, making use of the *Observer* pattern implementation provided in the Java platform. `ObservableSource` standardizes the process in which other classes access the values gathered from the hardware by providing a getter method with a known name and fulfilling the *Observer* pattern implementation.

```
package adas.ar.sensing;

public abstract class ObservableSource extends Observable {

  public Object getObject(){return null;}

}
```

5.1.3. Classes AgentComm and AgentNet

Both `AgentComm` and `AgentNet` classes

a) create a concrete instance of an `ObservableSource` class descendent: `ThreadDevice` or `ListenerDevice` (`ObservableSource` descendants connect and gather values from the hardware device) and

b) create a `PublisherThread` instance that publishes the gathered values.

```
package adas.ar.sensing;

public class AgentComm implements Agent {

  private ThreadPublisher threadPublisher;
  private ListenerDevice listenerDevice;

}
```

```
package adas.ar.sensing;

public class AgentNet implements Agent {

  private ThreadPublisher threadPublisher;
  private ThreadDevice threadDevice;

}
```

5.1.4. Class ThreadDevice

The `ThreadDevice` abstract class defines a contract for the classes that implement it to

a) deal with threading capabilities, by inheriting from the `Runnable` Java class; and
b) handle an execution thread, which make a network connection to a specific IP address and port.

The implementation of the thread functionality (i.e., the `run` method implementation) is done at the subclass of `ThreadDevice` defined for each device to be handled. ADAS provides `ThreadTobii` (for Tobii eye tracker) and `ThreadEmotiv` (for Emotiv EPOC headset).

```
package adas.ar.sensing;

public abstract class ThreadDevice
       extends ObservableSource implements Runnable {

  protected String ip;
  protected short port;

}
```

5.1.5. Class ListenerDevice

The `ListenerDevice` abstract class implements `SerialPortEventListener`, a class provided by the RXTX library that facilitates communication with serial ports. The `SerialPort` class from the `gnu.io` library stores a reference to a physical serial port. By implementing `SerialPortEventListener`, `ListenerDevice` is notified each time a value arrives to a pre-specified serial port in the computer. Then, the value can be gathered. For some devices, the values arrive to the port encoded as hexadecimal strings and need to be parsed. The `parse` method is responsible for decoding the data read from the port and transforming it into a value. The `parse` method is abstract in this class, and its implementation is done in subclasses of `ListenerDevice` defined for each device to be handled. ADAS provides `ListenerPosture`, `ListenerPressure`, and `ListenerSkin` subclasses, which provide parse method implementations for posture, pressure, and skin conductance sensing devices, respectively.

```
package adas.ar.sensing;

public abstract class ListenerDevice
       extends ObservableSource
       implements SerialPortEventListener {

  protected DelegatePerception delegatePerception;
  protected SerialPort port;
  protected abstract void parse(String line);

}
```

5.1.6. Class ThreadPublisher

The `ThreadPublisher` class

a) creates an execution thread implementing the `Runnable` Java interface;

b) implements `Observer`, and then has immediate access to gathered values from `ObservableSource` objects; and

c) uses the networking infrastructure provided by the Java platform within `ServerSocket` and `Socket` classes to deliver `Measure` instance objects via TCP connections. `ThreadSubscriber` instances make connections to the port used by `ThreadPublisher` to receive those `Measure` instances.

```
package adas.toolkit.net;

public class ThreadPublisher implements Observer, Runnable {

  private final int port;
  private ServerSocket listener = null;
}
```

5.2. Package ar.perception

The perception package, named `ar.perception`, encapsulates a hierarchy of classes that implements a set of perception algorithms whose purpose is to understand the signal values and acknowledge an affective state value. The `ar.perception` package is shown in Figure 5.2 using a UML class diagram. Red circles highlight classes previously mentioned.

The `DelegatePerception` instance provides `ThreadDevice` and `ListenerDevice` with perception algorithm implementation. It follows a *Delegate* pattern to connect with a perception algorithm provided in the perception package. The perception package is optional, since in several cases the perception algorithm is provided by the manufacturer of the hardware device as part of the external libraries. By default, the `DelegatePerception` instance is null.

The `DelegatePerception` interface defines the signature for the method `infer`. Classes that implement the `DelegatePerception` interface override the method `infer` and implement in it the perception algorithm. Instances of both the `ThreadDevice` and `ListenerDevice` classes call the method to invoke the inference process to be executed.

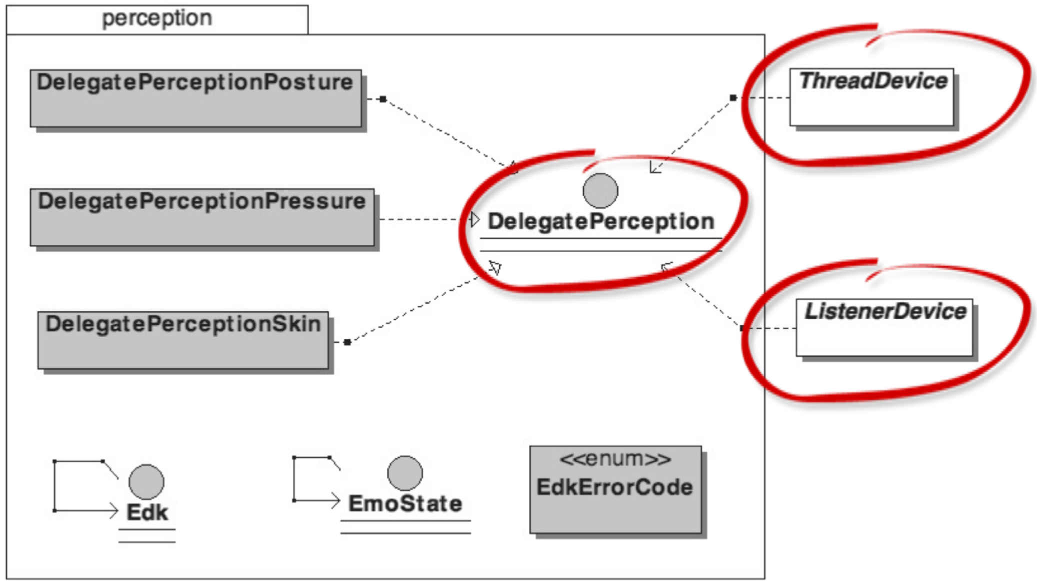


Figure 5.2. UML diagram of the package `perception`. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

ADAS includes implementation for:

a) The skin conductance sensor in the `DelegatePerceptionSkin` class. It filters, cleans, and normalizes the values gathered from the sensing process.

b) The pressure sensor in the `DelegatePerceptionPressure` class. It infers a frustration level from the values gathered from the sensing process.

c) The posture sensor in the `DelegatePerceptionPosture` class. It infers an interest level from the pressure values gathered from the sensing process.

No external SDK is required for these sensors but serial port communication capabilities are instantiated to work with this hardware. Algorithms for those sensors are implemented as described in Cooper et al. (2009).

No perception instances are provided by the agents dealing with the Tobii eye tracker, Emotiv, or MindReader because

a) access to Emotiv uses the interfaces `Edk` and `EmoState`, and the enumeration `EdkErrorCode`, which are wrapper classes for the Emotiv SDK in Java. They provide access to the native implementation of the SDK available for the Windows platform. Perception algorithms to infer excitement, engagement, boredom, meditation, and frustration are provided by the Emotiv SDK libraries.

b) access to the Tobii eye tracker is made using a Tobii SDK application coded in C++, streaming values into ADAS via a TCP/IP connection. Since the inference of affect from pupil dilation is an ongoing research topic, a conclusive algorithm is not defined in the framework.

c) access to MindReader is also achieved through a TCP/IP connection. No perception algorithm is required since MindReader reports values for agreeing, concentrating, disagreement, interested, thinking, and unsureness.

Developing additional classes that implement new inference algorithms, for instance, for customized algorithms that infer affect from Emotiv BCI raw data or from pupil size values, would create new `DelegatePerception` subclasses.

5.3. Package ar.integration

The hierarchy of classes for the integration package is depicted in Figure 5.3. A red circle highlights classes already mentioned. The design is explained in Chapter 4, and no further explanations are required aside from noticing that the generic class name `Delegate` in the design is changed here to `BlackboardFuseDelegate` to differentiate this class from other classes fulfilling the role of delegates.

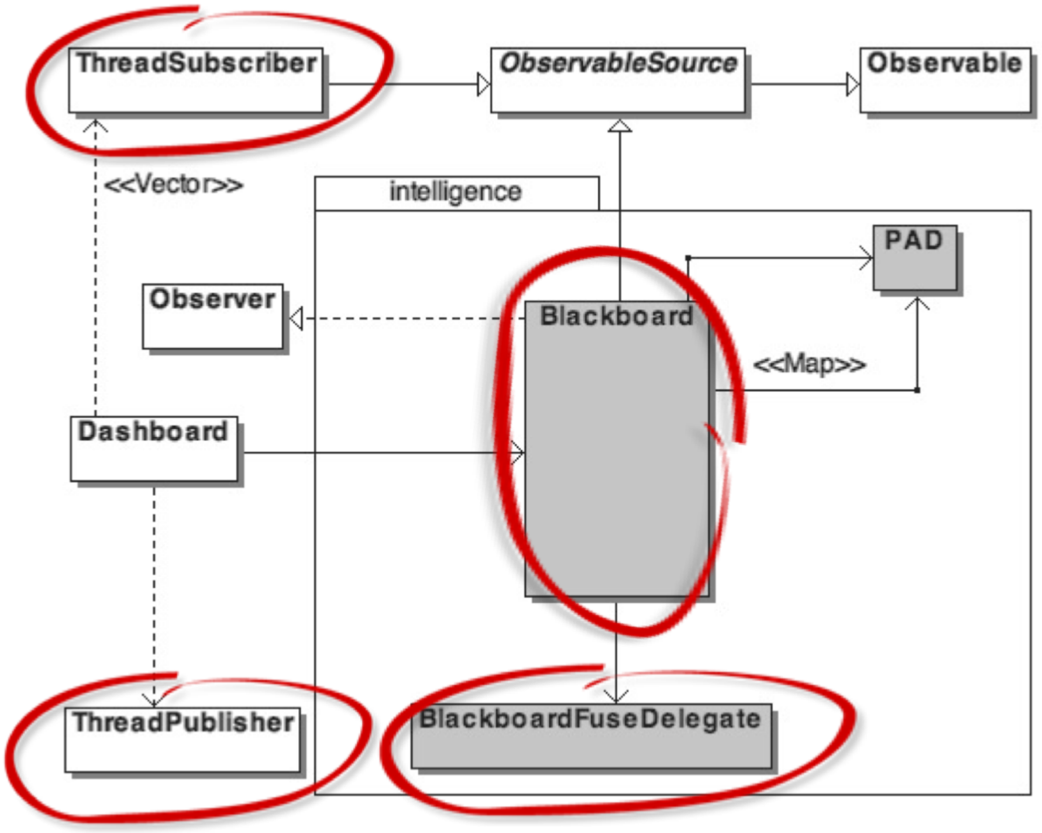


Figure 5.3. UML diagram of the package `integration`. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

5.4. Package toolkit

The `toolkit` package encapsulates a set of resources that support the functionalities of other packages. A hierarchy of classes and interfaces implements common tasks for networking, port handling, and data abstractions. To keep things in order, three sub-packages are defined for each set of tasks: `toolkit.net`, `toolkit.comm`, and `toolkit.data`.

5.4.1. Package toolkit.net

In the architecture, communication was the sole responsibility of the synapse component. The synapse component's implementation is absorbed by the `toolkit` package. The package `toolkit.net` implements distributed communication by defining data abstractions following the *Publisher-Subscriber* pattern. Figure 5.4 depicts its structure, which is summarized as follows:

a) The `ThreadPublisher` class defines an execution thread by implementing the `Runnable` interface from Java. It implements the `Observer` interface to make the data it gathers available to other classes, using an *Observer* pattern implementation. It receives connection requests via TCP/IP channels using an instance of the Java `ServerSocket` class, and each time a connection request is received `ThreadPublisher` creates an instance of `ThreadPublisherMinion` and assigns it to handle the connection.

b) The `ThreadPublisherMinion` class defines an execution thread by implementing the `Runnable` interface from Java. It sends the data being observed by the `PublisherThread` instance using a `Socket` instance.

c) The `ThreadSubscriber` class allows a connection to be established with `ThreadPublisher` instances. The `ThreadSubscriber` instances extend `ObservableSource`, and therefore the data gathered by them is shared using an *Observer* pattern implementation. Using a vector of `ThreadSubscriber` instances, the `ar.integration` package (particularly the `Dashboard` class) is able to collect data from a set of sources.

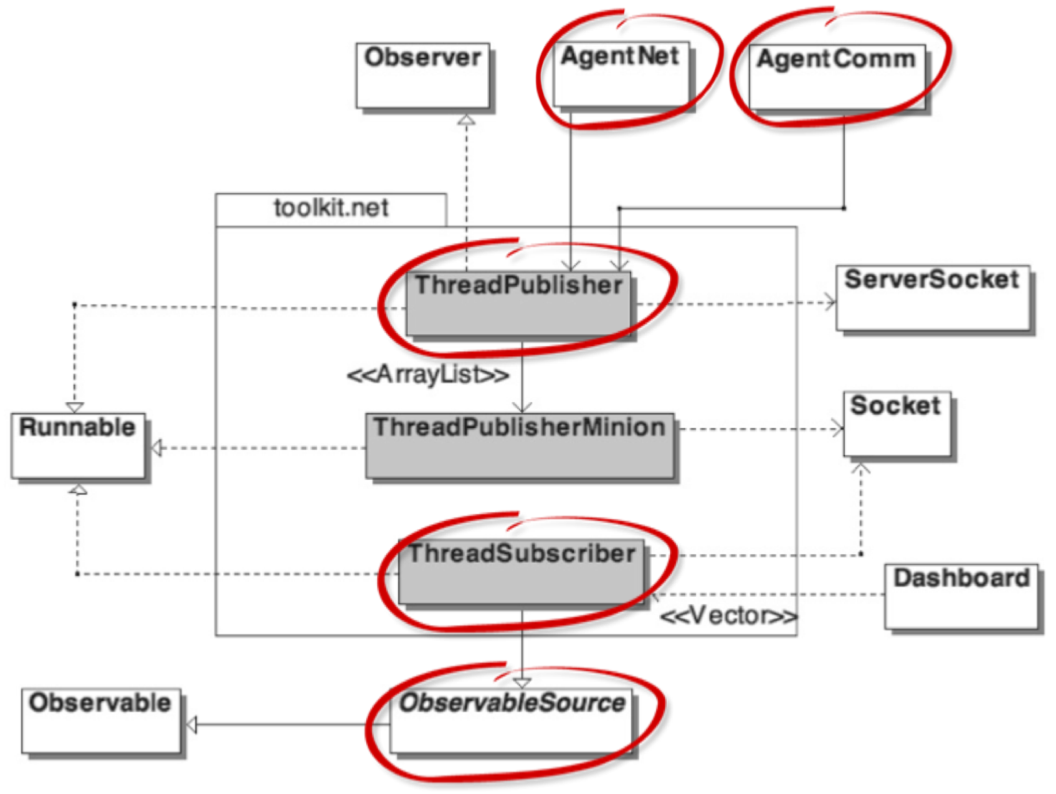


Figure 5.4. UML diagram of the package `toolkit.net`. In white (and when possible outside the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

5.4.2. Package toolkit.comm

Physical port reading is achieved with the support of a third-party Java library, RXTX. RXTX is a Java native library providing serial and parallel communication for the JDK on Windows and Linux platforms. ADAS uses RXTX version 2.1.7 (Windows distribution). The operations required for ADAS are encapsulated in the `SerialPortManager` class. It follows the *Singleton* pattern, and so the instantiation of `SerialPortManager` is restricted to one object; this guarantees that one object coordinates all actions across the system's physical ports. Figure 5.5 depicts its structure, which can be summarized as follows:

a) The `SerialPortManager` class allows the instantiation, as a *Singleton*, of an instance that scans all ports in the host computer and tries to create a connection with them. For the ports that accept a connection, it reads a buffer of data and checks whether the package format corresponds to known devices (those for posture, pressure, or skin conductance). A `ListenerDevice` instance is associated with those that fulfill the format requirements, and they are stored in a `Map` data structure. At this point of the implementation, adding automatic detection of more devices requires an update of the `SerialPortManager` class.

b) The `SerialDevice` class allows the storage of a port reference using a `SerialPort` instance (defined by the RXTX library) along with an attribute that identifies the device connected to that port (pressure, posture, or skin conductance).

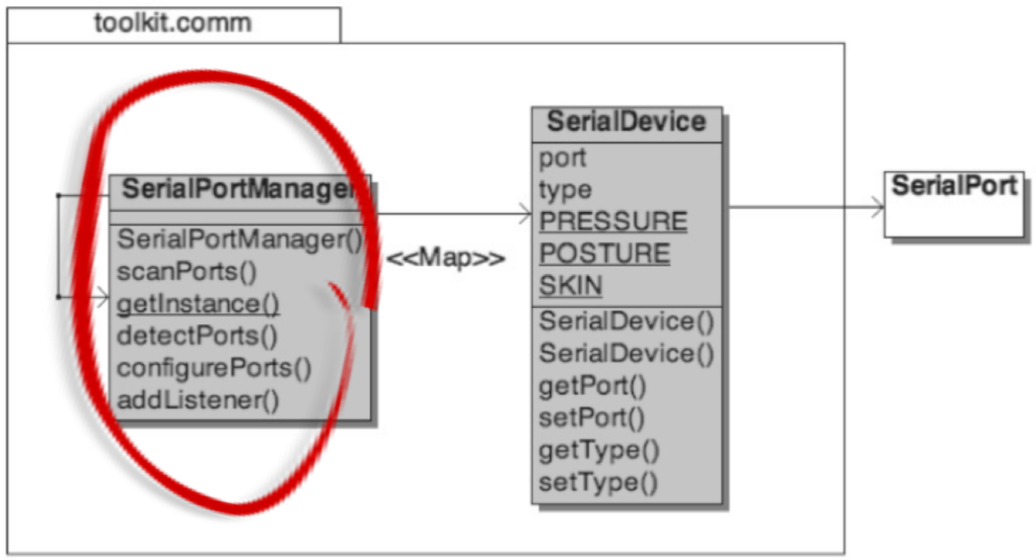


Figure 5.5. UML diagram of the package `toolkit.comm`. Attributes and methods are included. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

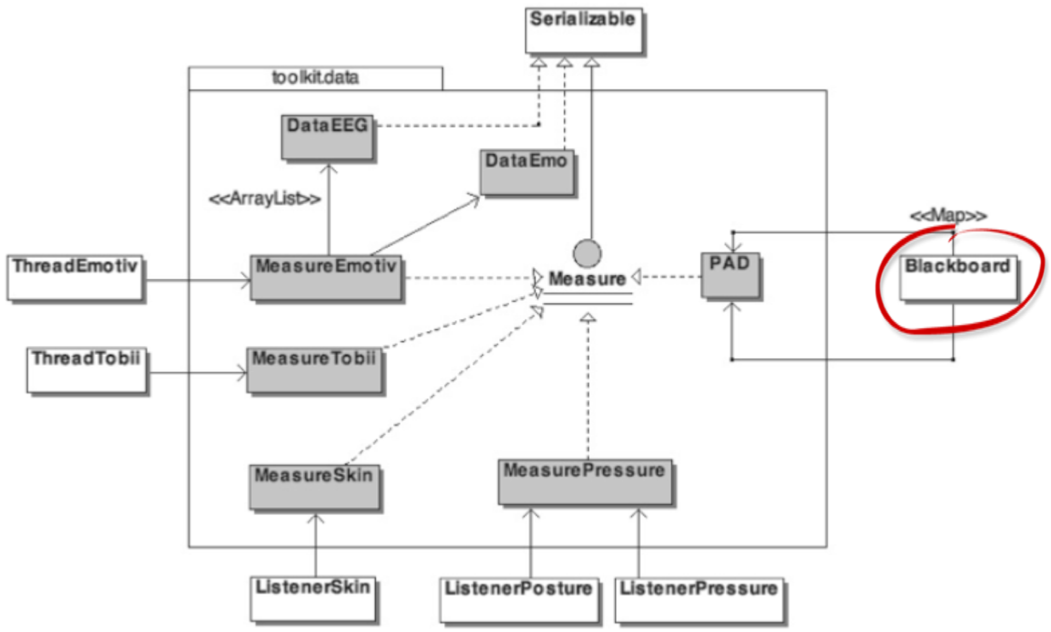


Figure 5.6. UML diagram of the package `toolkit.data`. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

5.4.3. Package toolkit.data

This package encapsulates data abstractions in a hierarchy of classes and one interface. This keeps the data and facilitates access to it. These classes prevent the data from being arbitrarily accessed. Figure 5.6 depicts its structure, which is summarized in the following subsections.

5.4.3.1. Interface Measure

After collecting the data from the device, parsing them, and applying perception algorithms if needed, the `ThreadDevice` and `ListenerDevice` instances encapsulate the resulting values in objects from classes that implement the `Measure` interface. The `Measure` interface

a) defines a contract that commit a class to provide an implementation for the method `toPAD`, which defines the algorithm for converting the values stored in the class to a PAD value; and

b) extends `Serializable` which allows serialization in instances that implement `Measure`. Thus, objects that implement the `Measure` interface are ready to be shared among Observers and Observables.

ADAS applies the `Measure` interface to the classes `MeasureEmotiv`, `MeasureTobii`, `MeasureFace`, `MeasureSkin`, and `MeasurePressure` (which works for both pressure and posture device readings). They define the attributes for data values gathered from a specific device:

a) The class `MeasureEmotiv` includes attributes for five emotions and a stack data structure for raw data samples. The class `MeasureEmotiv` is a composition of the

`DataEEG` and `DataEmo` instances, which store EEG raw data (14 channels) and Emotiv affective constructs, respectively.

b) The class `MeasureTobii` includes attributes for gaze points (x and y values) and pupil dilation.

c) The class `MeasureFace` includes six attributes for six emotions gathered from MindReader software.

d) The class `MeasureSkin` includes one attribute for arousal level.

e) The class `MeasurePressure` includes one attribute for frustration or interest. The `MeasurePressure` instances are used to store both posture and pressure data.

```
package adas.toolkit.data;

public interface Measure extends Serializable {

  public PAD toPAD();
}
```

The `Measure` interface extends the `Serializable` interface in Java, which enables the serialization of objects, i.e., the ability to represent objects as a sequence of bytes, including object data, information about the object type, and the type of data stored in the object. Serialization facilitates sending and receiving serializable objects following the *Publisher-Subscriber* pattern. Also, the `Measure` interface forces classes that implement it to define a method `toPAD()` that returns a `PAD` object equivalent to the `Measure` instance itself. The `Measure` interface is implemented by the classes `MeasureEmotiv`, `MeasureTobii`, `MeasureSkin`, and `MeasurePressure`.

5.4.3.2. Class PAD

The `PAD` class implements the `Measure` interface, and so `PAD` instances are serializable. `PAD` instances return a reference to themselves in their implementation of the `toPAD` method. The `PAD` class encapsulates pleasure-arousal-dominance equivalences of affect constructs defined by the devices handled in ADAS as constants. The `Blackboard` class (in the `ar.integration` package) creates a map data structure of `PAD` objects; this data structure has a location where each agent deposits the most recently collected data, expressed as a PAD value.

5.4.4. Package gui

Additionally, ADAS defines GUIs for the predefined agents, which are able to display, using plots and the CSV format, the data that is being collected and information about the status of the agent connected to the device being handled in real time.

The GUI structure follows the *Model-View-Controller* architectural pattern, where

a) the `gui` package defines the View;

b) the Model facade is the `ObservableSource` abstract class that connects with the View using an Observer-Observable structure; and

c) the controller is built over a listener structure on the panel's visual components.

The package's structure is depicted in Figure 5.7 for agents and Figure 5.8 for the central control unit, and described below:

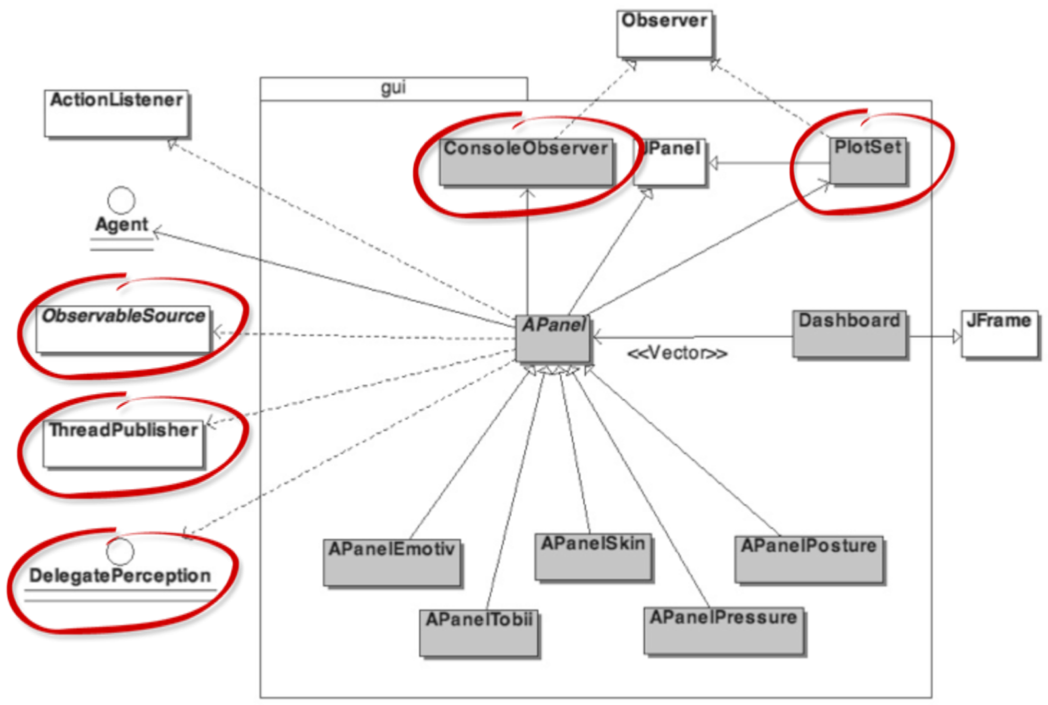


Figure 5.7. UML diagram of the package `gui`. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

a) The `Dashboard` class is an executable class. When run, it creates a desktop application that allows agents for each of the devices predefined in ADAS to be started. It puts one instance of each of the `APanel` subclasses in the same window, organized in vertical tabs. Each tab has in its label an on/off image that is updated to show whether or not the corresponding agent is running. The Figure 4.9 shows the GUI of the Dashboard application.

b) All the `APanel` subclasses (`APanelEmotiv`, `APanelTobii`, `APanelSkin`, `APanelPressure`, and `APanelPosture`) are also executable classes. This allows the running of agents individually. When executed directly, they create a window with the same content, even though the distribution of the graphical elements varies to make the GUI more user-friendly compared to the tabbed distribution in the Dashboard.

c) The `APanel` abstract class defines the underlying structure for the agent application. It (1) instantiates an `Agent` instance; (2) instantiates `ObservableSource` and `ThreadPublisher` instances, which will be passed to the `Agent` instance as parameter; and (3) optionally instantiates a `DelegatePerception` instance, which is passed to the `ObservableSource` instance (a `ListenerDevice` or a `ThreadDevice`) as a parameter. If a null parameter is received, the `ObservableSource` will create a `Measure` object with only the information gathered from the device. For instance, for the Emotiv EPOC headset a `DelegatePerception` is not needed since the Emotiv SDK reports emotion constructs directly; however, a `DelegatePerception` can be coded to implement a new algorithm to infer affect from raw brainwaves data. Others agents, such as those for Tobii, and skin, pressure, and posture sensing devices, require the coding of a `DelegatePerception` to implement the algorithm.

d) The `APanel` abstract class also defines the GUI components that are common to the application of all agents: (1) a `ConsoleOberver` instance that shows the data being gathered as a comma separated string and (2) a `PlotSet` instance that shows a Java Panel that creates a `PlotPanel` instance for each value to be shown, for instance, 29 are created in total for Emotiv. The `ConsoleObserver` and `PlotSet` classes implement the `Observer` interface; therefore, they are notified each time a new value is gathered in the `ObservableSource`, which, as shown before, extends the `Observer` class.

e) For the central control unit, whose functionality is implemented in the integration package, there is also a GUI `CentreDashboard`. The Figure 4.8 shows the GUI of the CentreDashboard application.

f) The `CubeCanvas` class extends the Java `Canvas` class. It defines a blank rectangular area of the screen onto which the application can draw PAD coordinate vectors as dots, and trap input events from the user, allowing the rotation and movement of the axis to facilitate the exploration and visualization of the cloud of cumulated PAD coordinate vectors.

g) The `PlotSetPAD` class is a container for three `Plot` instances that show the pleasure, arousal, and dominance values in plots. It is allocated in the second tab of `CentreDashboard` GUI.

h) The `SimulatorThread` class is provided as an auxiliary tool. It generates random PAD values. Thus, it is possible to play with the `CentreDashboard` application with no agent running, i.e., without using any device. This also allows the testing of the adaptation logic and functional logic.

5.5. Package adaptation

The `adaption` package hides and allows the extension of simple to complex decision processes for behavior-definition handling that defines whether or not an adaptation should be triggered, and if so which adaptation. It structure is shown in Figure 5.9.

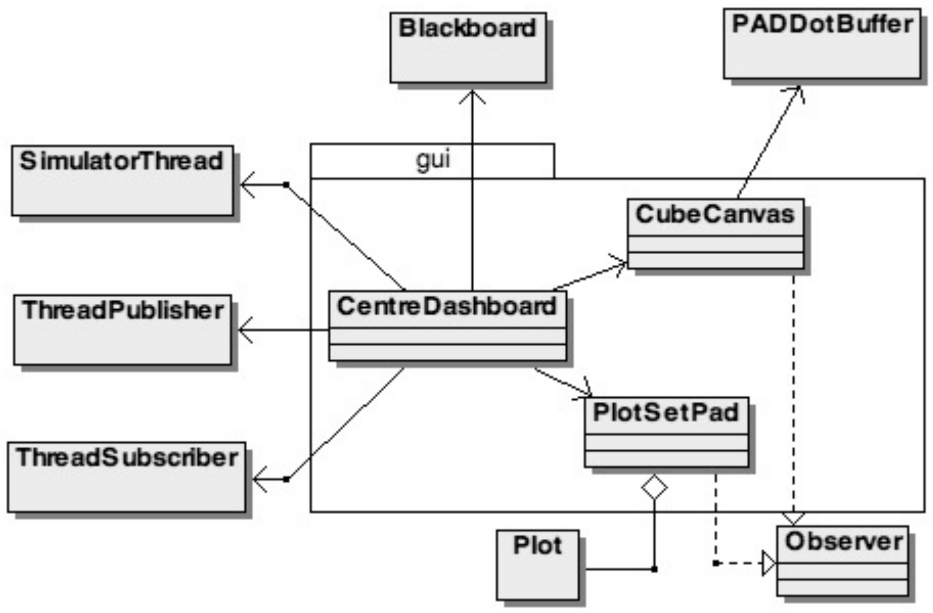


Figure 5.8. UML diagram of the package `gui.centre`. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

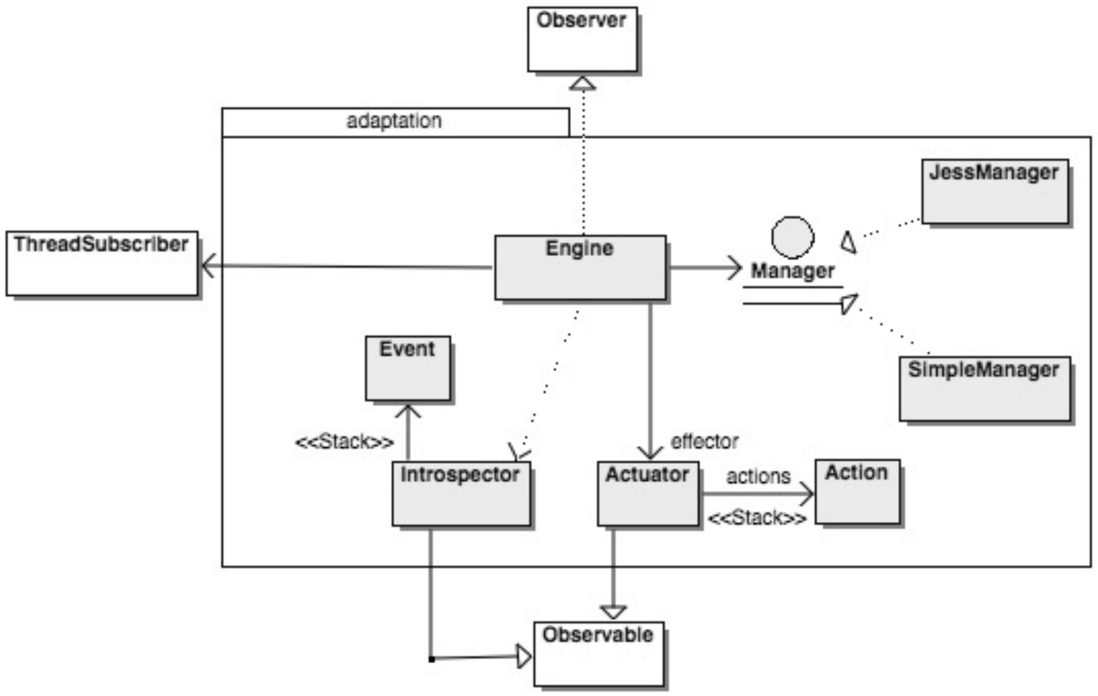


Figure 5.9. UML diagram of the package `adaptation`. In white (and when possible outside of the package box) are the classes and interfaces that do not belong to the package but are included to show implementation relationships.

The core of the adaptation package is the `Engine` class which

a) uses a `ThreadSubscriber` instance to receive data from the affect recognition logic, following the *Publisher-Subscriber* pattern. Received data is encapsulated in `Measure` instances and reported to an `Engine` instance using the *Observer* pattern – `ThreadSubscriber` is observable and `Engine` implements *Observer*.

b) uses a `Manager` instance to support dealing with adaptation rules. The `Manager` interface, follows the *Delegate* pattern; it hides and allows the extension of simple to complex processes for behavior-definition handling. Developers extend the class to implement specialized functionalities. The `Engine` class has a `Manager` instance. It is a customizable component. ADAS includes two pre-built implementations of the `Manager` class: a simple one containing a set of conditional statements and a complex one using Jess rule engine.

c) uses an `Introspector` class to handle the processes for introspection. It is an optional component for systems that require context awareness. The `Introspector` class receives notifications about updates in the functional logic and makes them available to `Engine` class through an observer-observable connection – the `Introspector` class extends the `Observable` class. The functional logic creates an `Introspector` instance and reports packed information to it in `Event` instances. The `Event` class encapsulates the information that the functional logic wants to share with the adaptation logic.

d) triggers actions in the functional logic using the `Actuator` class. The `Actuator` class extends the `Observable` class, and thus can be observed by the functional logic.

Particularly, an and interface called `ActuatorListener` is defined. The functional logic can implement the `ActuatorListener` interface to listen for adaptation that are requested to be triggered. Adaptation requests are encapsulated as `Action` instances.

5.6. Probes and Effectors

Functional logic is the responsibility of the developers; the only intervention required by developers is to provide interfaces for extracting information (probes) and for allowing changing the system (effectors). Their implementation is shown in Figure 5.10 and described below.

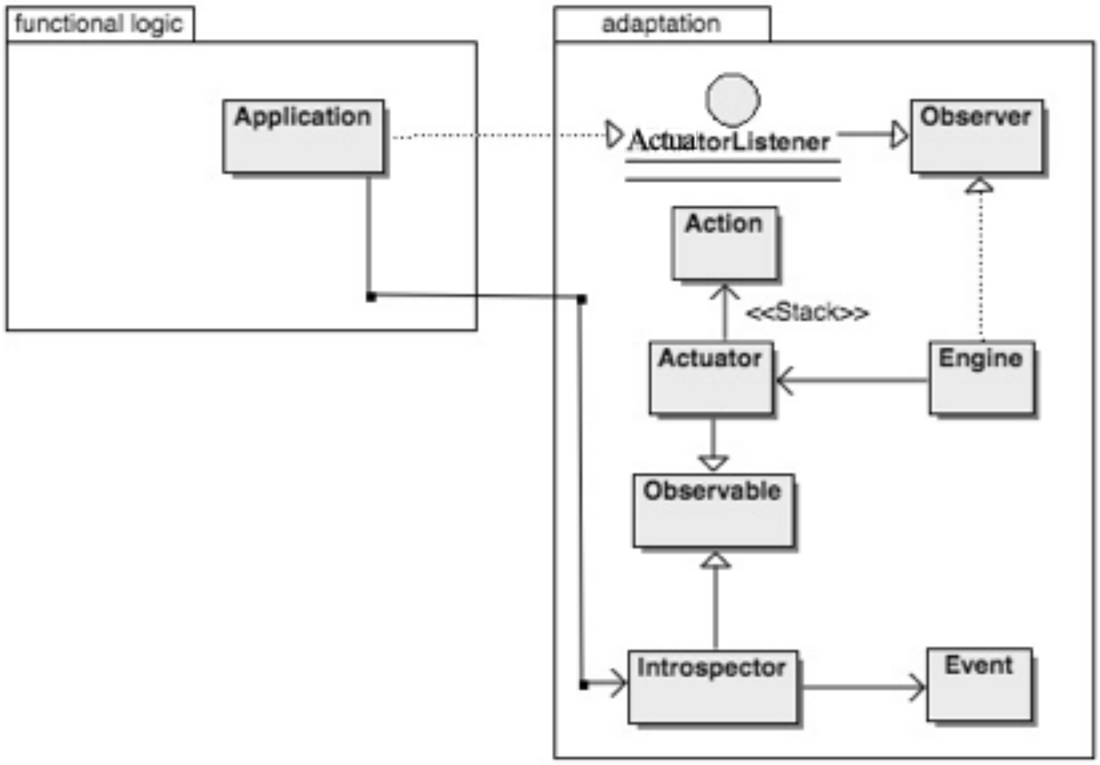


Figure 5.10. UML diagram for effector and probes. The effector is implemented using the *Observer* pattern between the functional logic and the `Actuator` class while probes are calls to an `Instrospector` instance method.

Effector connections are implemented using an implementation of the *Observer* pattern: the `Actuator` class is the Observable part and the functional logic implements the Observer with the `ActuatorListener` interface. Developers are required to implement, as part of the functional logic, a class that implements the `ActuatorListener` interface that provides a method `update()`. The method `update()` is automatically called from the `Actuator` class each time the adaptation logic wants to communicate a change to the functional logic. The method `update()` receives as a parameter the reference to the `Actuator` instance that contains the requested adaptation (encapsulated as an `Action` instance). The `Action` class is a container for a communication packet. It has two attributes: an integer value and a text string to be used as needed.

Probe connections are implemented using calls to an `Introspector` instance. The functional logic creates an `Introspector` instance and posts `Event` objects in it. The `Event` class is a container for a communication packet. It has two attributes: an integer value and a text string to be used as needed. The `Introspector` class implements the interface `Observer` and is observed by the `Engine` class.

The code below exemplifies an implementation of the functional logic with an effector and a probe in a class named `Application`.

```
public class Application implements ActuatorListener{

      private Introspector introspector;

      @Override
      public void update(Observable observable, Object arg) {

          Action actionToDo = ((Actuator) observable).getActions();
      }
      public void anyMethod() {

        instrospector.setEvent (new Event (1. "event name");
        ...
      }

}
```

5.7. Summary

In this chapter, I detailed the ADAS framework that empowers generality and cost-effectiveness by providing a well-understood set of assets for affect recognition and closed-loop adaptation. All the classes in each component are detailed in terms of object-oriented programming elements and connections. The implementation details described in this chapter are required when there is an intention to further extend or modify ADAS.

CHAPTER 6
PRODUCTION GUIDELINES

The third step in the SPL technical solution is the definition of production guidelines to explain how the components of the infrastructure can be integrated and customized. Johnson's (1997) statement that the best documentation for a framework should seem similar to a cookbook is taken as the foundation for these guidelines. First domain-specific patterns are used to reason about and organize the high level structures of the system that is to be developed; then, the system can be technically described using software design patterns and common object-oriented documentation techniques. I present a set of examples that teach developers how to manufacture a system. Three aspects of the production guidelines are described in the following sections:

a) the core component documentation describing how to run them and their basic operations at run time;
b) the optional and extensible component documentation, establishing the rules for their association, inclusion, and extension; and
c) the probe, gauge, and effector documentation.

The development task described assumes that the ADAS framework is installed on the developer's computer as described in Appendix A. Understanding the production process requires confidence in the meaning of the following terms: sensing, perception, integration, synapse, reaction engine, introspection, and behavior management. These have been described as steps of the affect recognition and adaptation processes. They have also been used as names for components and packages in order to relate abstract concept and development assets. They are documented in previous work as patterns and their relationships established using a pattern language structure (Gonzalez-Sanchez, Chavez-Echeagaray, Atkinson, & Burleson, 2012). They outline the production process as depicted in Figures 6.1., 6.2, and 6.3. The production process has four sequential stages, which are as follows: (1) establish affect recognition sources, (2) centralize affect recognition sources, (3) create the adaptation logic, and (4) connect adaptation with a functional logic. Stage 1 and 2 are related to building, extending, or reusing and affect recognition logic, and stage 3 and 4 are all about the adaptation and functional logics.

6.1. Establishing Affect Recognition Sources.

Creating an affect-driven self-adaptive system requires the use of sensing devices for the affect recognition process. At least one sensing device is required but more can be used. There are three possible circumstances that drive three possible sequences of actions in this stage, as observed in Figure 6.1. The result of this stage will be to have one or more agents running, each connected to a sensing device. Each agent streams affect measurements to a network port. The possible sequences are described in the following subsections and can be summarized as follows:

a) Sequence A: sensing devices to be used are one or more of those in the ADAS framework, described in Chapter 4. In this case, running the agent, provided in ADAS, completes stage 1. If the intention is to use the sensing device but to modify the ADAS agent for that device, follow sequence C.
b) Sequence B: a new type of sensing device is to be used, one that has inference algorithms embedded or has inference algorithms provided as part of its platform; so that connecting to the device provides affect measurements directly. In this case,

develop a new agent and implement the sensing process. Sensing is a core functionality in the agent.

c) Sequence C: a new type of sensing device is to be used, one that requires developers to implement an inference algorithm to transform the signals gathered into affect measurements. In this case, proceed as in sequence B and also implement the perception process.

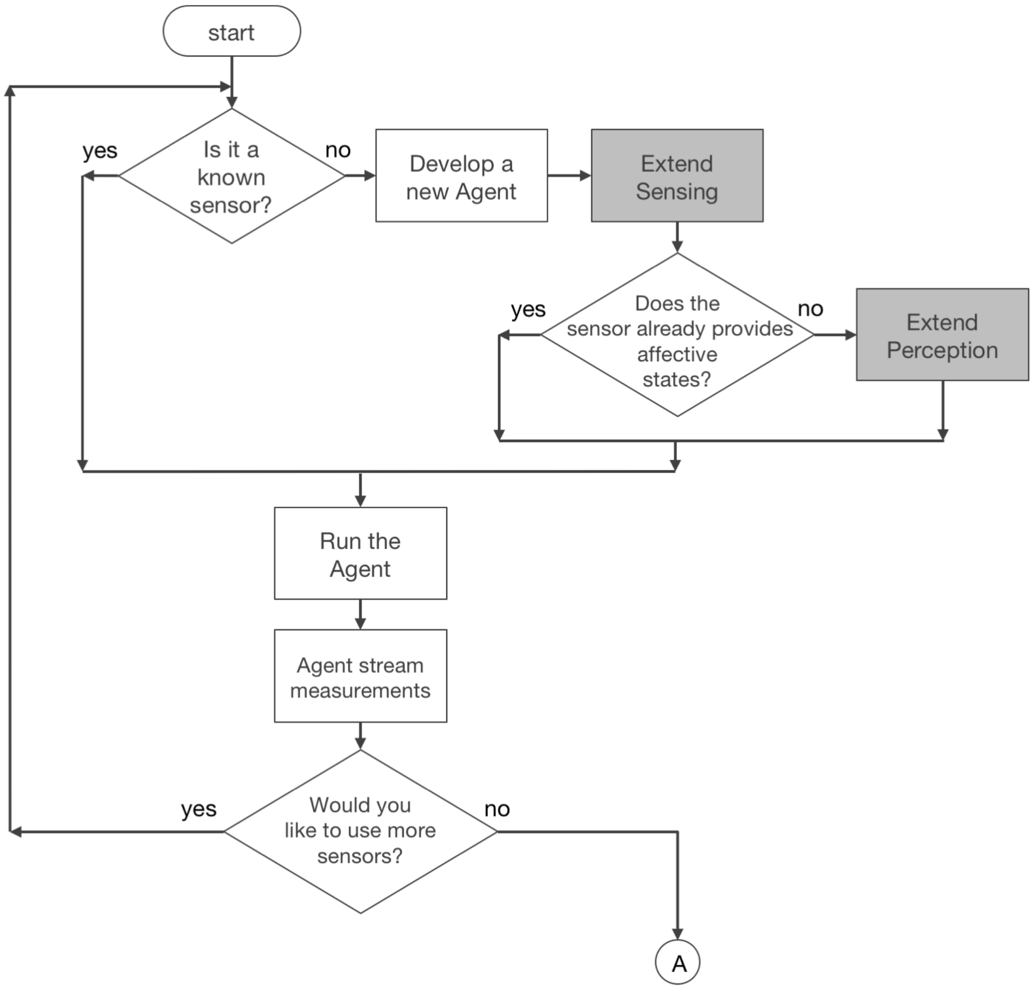

start
yes
Is it a known sensor?
no
Develop a new Agent
Extend Sensing
yes
Does the sensor already provides affective states?
no
Extend Perception
Run the Agent
Agent stream measurements
yes
Would you like to use more sensors?
no
A

Figure 6.1 Production process stage 1: establishing affect recognition sources.

6.1.1. Sequence A: Using ADAS Agents

Follow these steps to execute the ADAS built-in agents:

a) From the set of known devices, described in section 2.2, select the device to be used to collect data. Attach the device to the host computer and set up as needed.

b) Execute the agent that corresponds to the selected device. The agent GUI guides the next sequence of steps. The GUI will visualize the gathered data in plots and as CVS strings.

c) The data that is plotted and listed in the console is also streamed in real-time to a port in the host computer using a TCP/IP protocol. The list of ports used by each agent is presented in Chapter 4.

d) A TCP/IP client can access the affective data streamed by the agent. Implementing it is a standard procedure in most programming languages. Serialized objects of type `Measure` are streamed from the agent.

6.1.2. Sequence B: A New Type of Agent

Adding a new device to ADAS requires the steps described below to be followed. It assumes knowledge of the internal ADAS structure as described in Chapters 4 and 5. An agent for handling a hypothetical heart rate monitor is used as an example.

Class MeasureHeart

a) Create a new `MeasureHeart` class that must implement the interface `Measure`. This enables the serialization of the `MeasureHeart` class.

b) Inside `MeasureHeart` class, attributes should be declared to store the values provided for the device. For a heart rate monitor, a single attribute, `rate`, is enough.

c) Implementing `Measure` requires the method `getPAD` to be overridden. The method `getPAD` returns a PAD vector of [0, f(rate), 0], since heart rate is related to arousal, as a function of rate, but pleasure and dominance cannot be inferred from heart rate.

d) The array `LABELS` provides the names to be used for GUI elements when plotting variables stored in this class. For this example, there is only one: rate.

e) Implementing `Measure` also requires the method `getValuesFor` to be overridden.

```
public class MeasureHeart implements Measure {

  private double time;
  private float rate;

  public MeasureHeart(double time, String data)
    throws Exception {
    this.time = time;
    parse(data);
  }

  protected void parse(String dataIn) throws Exception {
    value = // convert the string read from the device in a float
  }

  public static final String[] LABELS = {"heart rate"};

  @Override
  public PAD toPAD() {
    PAD pad = new PAD();
    pad.p = 0;
    pad.a = value;
    pad.d = 0;
    return pad;
  }

  @Override
  public ArrayList<Float> getValuesFor(int i) {
    ArrayList<Float> result = new ArrayList();
    if (i == 0) {
       result.add((float) value);
    }
    return result;
  }

}
```

Class ThreadHeart

Create `ThreadHeart` and associate it with the SDK of the device. Get familiar with the SDK documentation and connect it by calling the SDK methods in this class. Usually, these are inside the `run()` method that must be included since the class implements `Runnable`.

a) Override the `getObject()` method.

b) Override the `run()` method. The method calculates a timestamp for the data. In this method the connection with the hardware device should be implemented. In the example below, no real connection is implemented and a random value provided as the value for the measure.

c) For educational purposes, an auxiliary method `createAndNotify()` is defined to create `Measure` and notify observers.

```
public class ThreadHeart extends ThreadDevice {

  private MeasureHeart measureHeart;

  public ThreadHeart (DelegatePerception dp) throws IOException {
    super();
    this.delegatePerception = dp;
  }

  @Override
  public Object getObject() {

    return measureDummy;
  }

  private void createAndNotify (double timestampsystem, double a) {

    measureHeart = new MeasureHeart(timestampsystem, a);
    setChanged();
    notifyObservers();

  }
@Override
  public void run() {
    stop = false;
    // time
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.HOUR_OF_DAY, 0);
    calendar.set(Calendar.MINUTE, 0);
    calendar.set(Calendar.SECOND, 0);
    long initsystem = calendar.getTimeInMillis();
    double timestampsystem = 0;
    // value
    while (!stop) {
      timestampsystem=
       (System.currentTimeMillis()- initsystem) * .001;
      createAndNotify (timestampsystem, Math.random());
      try {
        Thread.sleep(pause);}
      catch (
        InterruptedException ex) {
      }
    }
    Console.getInstance().log("ThreadDummy:run | done.");
  }

}
```

Class AgentHeart and APanelHeart

Create an agent, associate `PublisherThread` and `ThreadHeart` with it, and run it:

```
public class AgentHeart {

  main() {

    ThreadHeart threadHeart = new ThreadHeart(null);
    ThreadPublisher threadPublisher =
      new ThreadPublisher(7070, ThreadPublisherMinion.TYPE_OBJECT);
    agent = new AgentNet(((ThreadDevice) td), tp);
  }

}
```

Optionally you can add GUI elements inherited from `JFrame`, using some of the widgets provided in ADAS. The `init()` method is defined in `APanel` and it creates the `Agent` instance with the provided `PublisherThread` and the `ThreadHeart`.

```
public class APanelHeart extends APanel {

  public APanelHeart(boolean console) {
    ThreadHeart threadHeart = new ThreadHeart(null);

    ThreadPublisher threadPublisher =
      new ThreadPublisher(7070, ThreadPublisherMinion.TYPE_OBJECT);

    init(
      MeasureHeart.LABELS, threadHeart,
      threadPublisher, indicator, label );

    createGUI(console);
  }


  private void createGUI(boolean console) {
    // bar
    JPanel bar = new JPanel(new BorderLayout());
    bar.add(new JLabel("
      Publishing at 7979 | Mode: OBJECT"), BorderLayout.WEST);
    bar.add(buttonConnect, BorderLayout.EAST);
    bar.setBackground(Color.white);
    // connect
    JPanel connectionPanel = new JPanel(new BorderLayout());
    connectionPanel.setBackground(Color.white);
    connectionPanel.add(bar, BorderLayout.EAST);
    createGUI(connectionPanel, console);
  }

  public static void main(String[] args) {
    JFrame win = new JFrame("Agent | Dummy");
    // window
    APanelHeart md = new APanelHeart(true);
    win.setLayout(new GridLayout(1, 1));
    win.add(md);
    win.addWindowListener(new java.awt.event.WindowAdapter() {
      @Override
      public void windowClosing(java.awt.event.WindowEvent e) {
        md.close();
        System.exit(0);
      }
    });
    win.pack();
    win.setSize(400, 600);
    win.setVisible(true);
  }
}
```

6.1.3. Sequence C: A New Type of Agent with a New Inference Algorithm

Proceed as in section 6.1.2 to create a new type of agent. Then write your inference algorithm implementing a `DelegatePerception`. and overriding the method `infer`. The method `infer` receives an object `Measure` with the data sensed as a parameter. I define a simple one below for demonstration purposes:

```
package adas.ar.perception;

public class DelegatePerceptionHeart
  implements DelegatePerception {

  @Override
  public Measure infer(Measure m) {

     // implement here the inference process using m.
     // store the result in m itself.
     // update the class MeasureHeart to add attributes as needed.
     return m;
  }
}
```

Then modify the class `ThreadHeart` to add a call the inference algorithm:

```
public class ThreadHeart extends ThreadDevice {

   //...

  private void createAndNotify (double timestampsystem, double a) {

    MeasureHeart mh = new MeasureHeart(timestampsystem, a);
    measureHeart = DelegatePerceptionHeart.infer(mh);

    setChanged();
    notifyObservers();
  }
}
```

6.2. Centralizing Affect Recognition Sources.

Once the sensing devices and agents are running, the next step is to decide whether the central control unit will be used. There are three possible circumstances that drive three possible sequences of actions in this stage, as observed in Figure 6.2. The sequences are as follows:

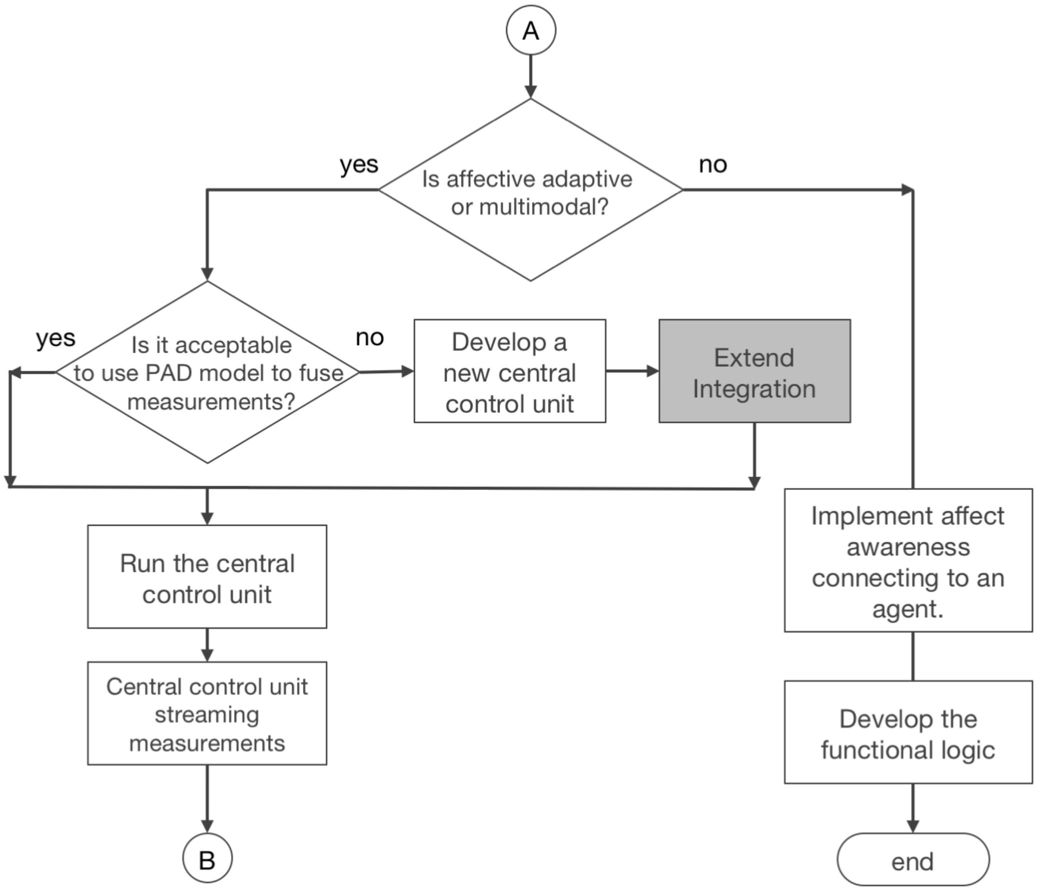

A
Is affective adaptive or multimodal?
yes
no
Is it acceptable to use PAD model to fuse measurements?
yes
no
Develop a new central control unit
Extend Integration
Run the central control unit
Central control unit streaming measurements
B
Implement affect awareness connecting to an agent.
Develop the functional logic
end

Figure 6.2 Production process stage 2: centralizing affect recognition sources.

a) Sequence A: monomodal affect recognition for affective awareness, as described in section 3.5.1. and shown in Figure 3.8, does not require the central control unit. The central control unit is only required for multimodal affect recognition (synchronizing and fusing measurements) and to set an emotional state model for the measurements.

b) Sequence B: developers agree to use PAD emotional state model, addition of PAD vectors as a fusion technique and a state-machine synchronization approach. In this case, use the central control unit provided in the ADAS framework. If a new agent was created and is being used, proceed as in sequence B. It will not be necessary to define a `BlackboardFuseDelegate` but the IP address to which the new agent streams its information needs to be set up.

c) Sequence C: a new type of fusion or synchronization approach is to be implemented. In this case, develop a new central control unit and implement integration (fusion and synchronization processes). Integration is a core functionality in the central control unit.

6.2.1. Sequence A: Monomodal Affect Awareness without a Central Control Unit

Implement your own customized functional logic. An implementation in Java of the code in the target system reading data from the agent is as follows:

```
...
Measure measure

Socket client = new Socket(InetAddress.getByName(Ip), port);
ObjectInputStream ois=
  new ObjectInputStream(client.getInputStream());
while (!stop) {
  ...
  measure = ois.readObject();
  ...
}
```

But also, loading `adas.jar` in the target application, ADAS classes can be used to read data from the agents instantiating a `ThreadSubscriber` object. Only one parameter is need to specify a period time for the thread to suspend execution in order to make processor time available to other threads of the application or other applications that might be running on a computer system.

```
  private ThreadSubscriber subscriberThread =
    new ThreadSubscriber(100);
```

Then the IP and port to which the `ThreadSubscriber` instance connects are specified. When specifying the port, a second parameter defines the type of information that is expected, CSV values or serialized Java objects using `ThreadSubscriber` class public static variables `TYPE_STRING` and `TYPE_OBJECT`, respectively. Then the thread can be executed and an Observer added.

```
public class Demo implements Observer {

  public Demo() {
    subscriberThread.setIp("localhost");
    subscriberThread.setPort(7474, ThreadSubscriber.TYPE_STRING);
    Executors.newCachedThreadPool().submit(subscriberThread);
    subscriberThread.addObserver(this);
  }

  private void close() {
    subscriberThread.stop();
  }

  public static void main(String[] args) {
    Demo tester = new Demo();
  }

  @Override
  public void update(Observable o, Object arg) {
    System.out.println( ((ThreadSubscriber) o).getObject() );
  }
}
```

6.2.2. Sequence B: Using ADAS Central Control Unit

This scenario has no adaptive capabilities and a multimodal affect recognition capability. It executes a set of agents and a central control unit that synchronizes and fuses the multimodal inputs. Follow these steps to execute one of the built-in agents:

a) Execute the agents that correspond to the selected devices in the computer(s) hosting the devices. The GUI of each agent guides the next sequence of steps. The GUI will visualize the gathered data in plots and as CVS strings. If you plan to use a single computer for all devices then execute `dashboard_agents.bat`. This will allow you to run and stop agents from a single GUI. If not, run the BAT file that corresponds to the required agent in each host computer.

b) The data that is being plotted and listed in the console is also streamed in real time to a port in the host computer using a TCP/IP protocol. The list of ports used by each agent is presented in Chapter 4.

c) Execute the central control unit. There are two options for this step: (1) if a single computer is used for all agents, then run the central control unit in that host, executing `dashboard_centre.bat` and using the list of default ports available in Chapter 4; or (2) run the central control unit in one of the host computers, either one of the computers running agents or an additional computer, by executing `dashboard_centre.bat` there. In this case, a file with the address of the computers running the agents should be provided and they should be accessible from the central control unit host, as in the box below, where the file `agents-ipaddress.txt` contains the address and port to which the agents are streaming data. For example,

```
emotiv 148.202.3.55 1407
tobii  148.202.3.10 1401
mouse  148.202.3.75 1405
chair  148.202.3.44 1404
```

The result of this stage will be to have the central control unit running and connected to agents and each agent connected to a sensing device. The agents and the central control unit streams affect measurements to a network port.

6.2.3. Sequence C: Customizing the Central Control Unit

Optional and variant features of the infrastructure are available in the central control unit of the affect recognition logic (integration and synapse). Optional capabilities are described here, as agents can be used without running a central control unit. Variant capabilities for these components can be handled as described below.

An extension point is in the integration component, where the fusion process in multimodal approaches is customized using a delegate. The extension entails the following steps:

First, create a new class that extends the `BlackboardFuseDelegate` class, and override the method `fuse()`. This method returns a PAD value and receives a reference to a map with the PAD values to be fused. The original `fuse()` method in the `BlackboardFuseDelegate` class uses vector addition as a strategy to fuse values:

```
public class NewFuseDelegate extends BlackboardFuseDelegate {
    @override
    public PAD fuse (Map <String, PAD> values) {
      synchronized (values) {
        PAD pad = new PAD();

        // sync and fuse algorithms here!

        return pad;
      }
  }
```

Second, when creating a `Blackboard` instance, assign an instance of the new class to be used by the `Blackboard` instance for fusing values:

```
  Blackboard blackboard = new Blackboard(new NewFuseDelegate());
```

Finally, instantiate a central control unit:

```
  public final class MyCentralUnit {

    // attributes

    private final ExecutorService service
      = Executors.newCachedThreadPool();

    private final ThreadPublisher publisherThread
      = new ThreadPublisher (7575, ThreadPublisherMinion.TYPE_OBJECT);

    private final ThreadSubscriber [] subscriberThread ;

    private final Blackboard blackboard
      = new Blackboard(new FuseDelegate());

    // main

    public static void main(String args[]) {

      MyCentralUnit cu = new MyCentralUnit();

    }
    // contructor
    public CentreDashboard() {

      // ...
      // sensors advocates

      for (int i=0; i<NUM_AGENTS; i++) {
        subscriberThread[i].setPort
          (sensor[i].port, ThreadSubscriber.TYPE_OBJECT);
        subscriberThread[i].addObserver(blackboard);
      }

      // source = blackboard
      blackboard.addObserver(publisherThread);

      service.submit(subscriberThread);
      service.submit(publisherThread);

    }

  }
```

6.3. Adaptation Logic

Once the affect recognition logic is running, the final step is to implement adaptation logic. There are three possible circumstances that drive three possible sequences of actions in this stage, as observed in Figure 6.3. Sequences are as follows:

a) Sequence A: customize a reaction engine for the system without introspection and without support for complex adaptation rules.

b) Sequence B: customize a reaction engine for the system with introspection but without support for complex adaptation rules.

c) Sequence C: customize a reaction engine for the system with introspection and with support for complex adaptation rules using a productions rule engine.

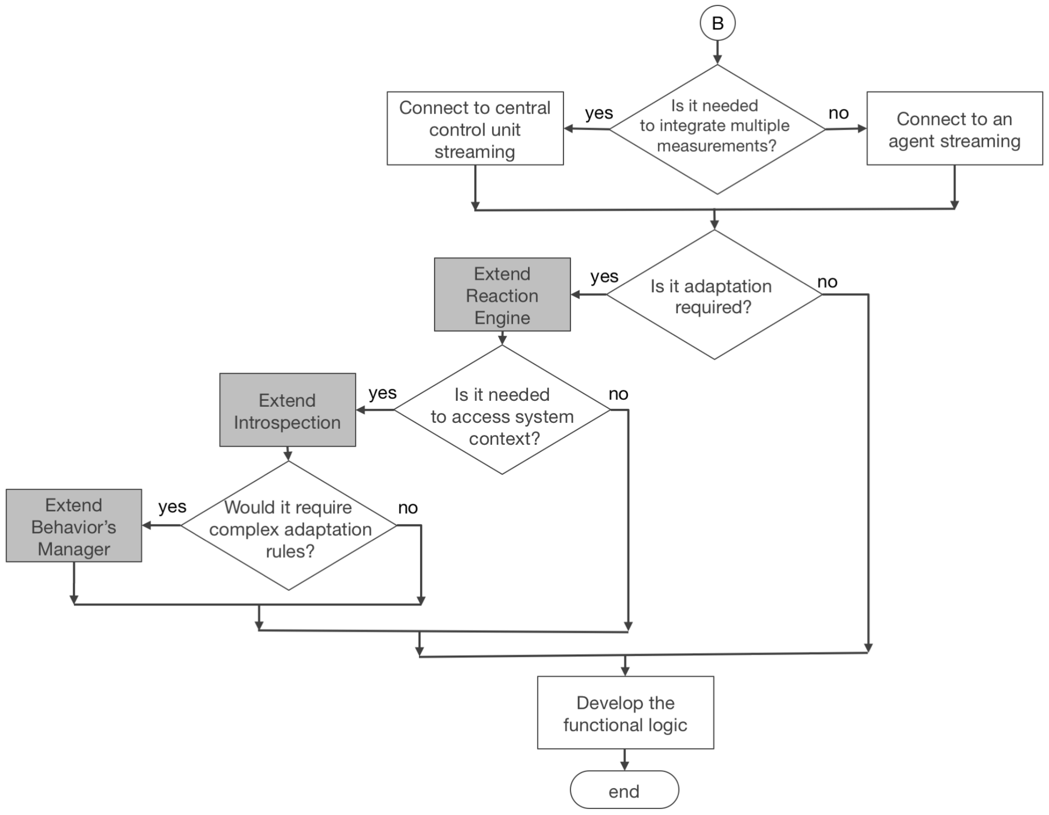

B
Is it needed to integrate multiple measurements?
yes
no
Connect to central control unit streaming
Connect to an agent streaming
Is it adaptation required?
yes
no
Extend Reaction Engine
Is it needed to access system context?
yes
no
Extend Introspection
Would it require complex adaptation rules?
yes
no
Extend Behavior's Manager
Develop the functional logic
end

Figure 6.3 Production process stage 3: adaptation and functional logics.

<u>6.3.1. Sequence A: Reaction Engine</u>

At this point implement your own affect aware functional logic. A TCP/IP client can be implemented to gather the data that any agent or the central control unit are streaming. When using the central control unit, as provided in ADAS, the obtained result is a PAD value in a `PAD` instance. When using agents, as provided in ADAS, the obtained result is a `Measure` objects or CSV strings. Implementing a TCP/IP socket is a standard procedure in most programming languages.

```
...
PAD pad;
Socket client = new Socket(InetAddress.getByName(Ip), port);
ObjectInputStream ois =
  new ObjectInputStream(client.getInputStream());
while (!stop) {
  ...
  pad = ois.readObject();
  ...
}
```

But also, loading `adas.jar` in the target application, ADAS classes can be used to read data from both agents and a central control unit instantiating a `ThreadSubscriber` object. The only difference is the port number to be specified. For the central control unit two ports are available: 7474 for CSV values and 7575 for serialized objects.

6.3.2. Sequence A: Reaction Engine and Introspection

Optional and variant features of the infrastructure are also available in the adaptation logic, specifically in the `Manager` interface. As described, adaptation is an optional capability and it is meant to handle variant scenarios. This section reviews the simplest scenario. Customization of the adaptation logic functionality is realized by following the *Delegate* pattern while implementing the `Manager` interface. The code below shows a simple implementation in a class named `ManagerSimple`.

```
public class ManagerSimple implements Manager {

  protected LinkedList<Measure> ms;
  protected LinkedList<Event> es;
```

```
    @Override
    public Iterator<Action> analyze (Observable o) {
      LinkedList<Action> a = new LinkedList<Action>();
      if (o instanceof Introspector) {
        Event e = ((Introspector)o).getEvents();
        System.out.println("new object E " + e);
        es.add( e );
      } else {
        Measure m = ((Measure)((ThreadSubscriber)o).getObject());
        System.out.println("new object M " + m);
        ms.add( m );
      }

      // ...calculate new action IF-ELSE ...
      a.add( new Action(id, name));
      return a.iterator();
    }

  }
```

This code:

a) implements the `Manager` interface;

b) declares containers (`LinkedList`) to store `Measure` and `Event` objects; and

c) overrides the method `analyze()`. This method executes each time an `Event` or `Measure` object is received by the `Engine` class. It receives an `Observable` object as a parameter, which represents an instance of `Introspector` or `ThreadSubscriber`; the first provides `Event` instances and the second `Measure` instances. Then actions can be calculated, in this case using nested if-else statements. Adaptations to be triggered are added to a list as `Action` instances, and finally an iterator to the list is returned.

6.3.3. Sequence A: Reaction Engine and Introspection with Production Rule Engine

This applies when the implementation of adaptive capabilities goes beyond simple conditional scenarios, for instance, when moving to software that has the capacity to "reason" using the knowledge supplied to it. External libraries can be connected to the infrastructure. This may require the use of rule engines. Rudolph (2003) suggests the following guidelines to decide when it might be valuable to use a rule engine:

a) the algorithm involves significant conditional branching or decision-making;
b) the rules are not static but likely to change over time due to the nature of the application; and
c) a convenient structure for separating "business logic" from the rest of the system is necessary, aiding in the effort to clearly "separate concerns".

As an example, I describe the steps to integrate Jess, a rule engine and scripting environment written in Java and created at Sandia National Laboratories. Using Jess, you can build Java software that has the capacity to "reason" using knowledge you supply in the form of declarative rules. Jess is small, light, and one of the fastest rule engines available.

Following the Jess documentation and assuming that Jess is installed as described there, embedding Jess with ADAS is relatively straightforward. An implementation of the `Manager` interface that integrates Jess is as follows:

a) Implement the `Manager` interface.
b) There is no declaration of local containers to store `Measure` and `Event` objects; instead they are added directly to the rule engine, which handles them.
c) Override the method `analyze()`. This method executes each time an `Event` or `Measure` object is received by the `Engine` class. It receives an `Observable` object as a parameter, which represents an instance of `Introspector` or `ThreadSubscriber`; the first provides `Event` instances and the second `Measure` instances. Then actions can be

calculated, in this case using Jess. Adaptations to be triggered are added as `Action` instances to a list, and finally an iterator to the list is returned.

d) The constructor method in the `ManagerJess` class shows the inclusion of the Jess rule engine by instantiating the `Rete` class, which loads the rules from a text file named rules.clp. The `Rete` instance can then be repetitively used to process inputs with the loaded rules and to calculate actions to trigger.

```
public class ManagerJess implements Manager {

  private Rete jessEngine;


  public ManagerJess() {
    try {
      jessEngine = new Rete();
      jessEngine.reset(); jessEngine.batch("rules.clp");
    } catch (JessException je) {}
  }

  @Override
  public Iterator<Action> analyze (Observable o) {
    // receive Measure and Event instances - trigger Action
    Action a = null;
    if (o instanceof Introspector) {
      Event e = ((Introspector)o).getEvents();
      System.out.println("new E " + e);
      try { jessEngine.add( e );} catch (JessException ex) {}
    } else {
      Measure m = ((Measure)((ThreadSubscriber)o).getObject());
      System.out.println("new M " + m);
      try {jessEngine.add( m );} catch (JessException ex) {}
    }



    // calculate new action JESS
    try {
      jessEngine.run();
    } catch (JessException ex) {
    }
    return jessEngine.getObjects(new Filter.ByClass(Action.class));
  }
}
```

In Jess, every rule has a name, an optional documentation string, some patterns, and some actions. A pattern is a statement of something that must be true for the rule to apply. An action is something the rule should do if it does apply, for instance, a rule for the following statement is shown below:

"If boredom, send the user a wake up action; but only if a UI event was also reported in the last time window unit."

```
(defrule boredom-wakeup
    "boredom -P-A-D then do something."
    (Request {quantityEvent >= 0})
    (Measure {pleasure < 0 && arousal < 0 && dominance < 0})
    =>
    (add (new Action "wake up the user")))
```

6.4. Functional Logic and their Probes and Effectors

The last stage is to implement the system itself, providing a functional logic with effectors, and to include probes if required. A functional logic can be written as follows:

a) If system context is needed, create an `Introspector` instance. It is used to send notifications of the system's event to the adaptation logic. In the example, random events are sent in a loop in method `doIt()`.

b) Create an `Engine` instance. The following parameters are passed to this instance: (1) an `EffectorListener` that will be notified of adaptation actions to be performed; (2) the `Introspector` instance used to add `Events`; (3) an `Actuator` instance, which will be linked to the `EffectorListener`; and (4) a `Manager` delegate, in this case an instance of the `ManagerSimple` class created above.

c) Implement the `EffectorListener` interface and override the method `update()`. This method is called each time that the `Manager` infers that a new adaptation must be performed. The code shows how `Action` objects are obtained. In the example, the `Action` instance is stored in the attribute named `action` and used in the loop inside the `doIt()` method.

d) Define the functional logic for the system. In this example a dummy functional logic is coded in the method `doIt()`, generating random `Event` instances and handling `Action` instances as adaptation requests.

```
public class FunctionalLogic implements EffectorListener {

  private Introspector introspector;

  private Action action;

  private Engine engine;




  public FunctionalLogic() {

    introspector = new Introspector();
    engine = new Engine(this, introspector,
                        new Actuator(), new ManagerSimple());
  }

  public void doIt() {

    int i = 0;
    while (true) {
      introspector.setEvent(new Event(i++, "a new event"));
      if (action != null) {
        System.out.println ("New Action to be done: " + action);
        action = null;
      }
      try {Thread.sleep(10000);} catch (InterruptedException ex) {}
    }
  }
```

```
    @Override
    public void update(Observable observable, Object arg) {

      action = ((Effector) observable).getActions();
    }

    public static void main(String [] args) {

      FunctionalLogic f = new FunctionalLogic();
      f.doIt();
    }

  }
```

6.5. Summary

In this chapter, the production guidelines are described, defining four stages and the sequences of actions to be chosen in each stage: stage 1 describes the selection of affect recognition sources ranging from using a predefined set of sensors and inference algorithms to extend infrastructure for new sensing devices and for new inference algorithms; stage 2 is about choosing whether to use monomodality, multimodality using the provided synchronization and fusion approaches, or multimodality extending the infrastructure with new synchronization or fusion approaches; and stage 3 describes the adaptation logic creation with options ranging from a simple adaptation logic without introspection, a simple adaptation logic with introspection, and an adaptation logic with introspection and complex adaptation rules using a production rule engine.

This chapter examined the manner in which production guidelines enable feasibility and cost-effectiveness by helping engineers understand the fine-grained details of particular affect and adaptation concepts as operational capabilities and by supporting their implementation or extension using the ADAS framework. Three aspects of the production guidelines are described: (1) the core component documentation, providing instructions for their use; (2) the optional and extensible component documentation, establishing the rules for their association, inclusion, and extension; and (3) the probe, gauge, and effector documentation.

CHAPTER 7
CASE STUDIES

This chapter introduces case studies implementing affect recognition, affect awareness, and affect-driven adaptation in a number of systems. Case studies were run to evaluate the operation of the proposed SPL technical solution, ADASPL. Diverse systems were developed adopting ADASPL, incrementally increasing in complexity and number of elements. These represent an in-depth demonstration of ADASPL that provide evidence (discussed later) for strength of and benefits of adopting this approach.

Gomaa (2004) stated that a minimum of three systems are required for the evaluation of an SPL. Accordingly, I evaluated ADASPL using 11 systems: 9 systems using ADASPL; and 2 systems using alternative approaches for comparative purposes. The scenarios covered go from partial or full use of the affect recognition logic to affect-driven adaptation with conditional rules or fuzzy rules. Particularly relevant are two versions of a real-world system: one version using ADAS and another not using it. The systems incrementally increase in complexity, integrating more core, optional, and variable components. Moreover, two implementations of affect recognition logics were built from scratch and compared with ADAS affect recognition. One set of systems was developed by undergraduate students participating in capstone projects (C1-C8) and another set by staff of research groups (R1-R3). Tables 7.1 and 7.2 summarize each set, respectively. These sample applications serve to demonstrate the applicability of the proposed approach across two of the three fields of interest, while also demonstrating the feasibility of broader deployment.

Table 7.1. Summary of Case Studies Developed as Capstone Projects

ID	Name	Adaptation	Affect recognition	Status	Dates	# of people	Language
C1	AAE	NO	from scratch (multimodal)	OK	S13-F13	3	Java
C2	Companion 2D	from scratch	ADAS (monomodal)	Incomplete	S13-F13	4	Java
C3	Pac-Man	from scratch	ADAS (multimodal)	OK	F13-S14	5	Java
C4	Horror Game	from scratch	ADAS (multimodal)	OK	S14-F14	5	C++
C5	Memory Game	from scratch	ADAS (multimodal)	OK	F14-S15	6	C#
C6	Shooting Game	from scratch	ADAS (multimodal)	Incomplete	F14-S15	5	C++
C7	Companion 3D	from scratch	ADAS (multimodal)	Discarded	F14-S15	4	Java
C8	Videos	NO	ADAS (multimodal)	Discarded	S15-F15	4	Java

Table 7.2. Summary of Case Studies Developed by Staff of Research Groups

ID	Name	Adaptation	Affect recognition	Status	Dates	# of people	Language
R1	Virtual World	NO	ADAS (monomodal)	OK	S13	3	Java
R2	AMT	from scratch	from scratch (multimodal)	OK	S09-F12	15	Java
R3	Enhanced Affective Tutor	ADAS	ADAS (multimodal)	OK	S15-F16	3	Java

7.1. Participants

To evaluate ADASPL, teams of developers were recruited and asked to build affect-driven adaptive systems; they had the freedom to define the functionality and adaptation goals but the domain was limited to learning and gaming, and they were asked to primarily address the concern of improving user experience. Case studies were run with:

a) research assistants in the Advancing Next Generation Learning Environments (ANGLE) Lab.
b) undergraduate students involved in one-year capstone projects. The capstone project courses (two semesters long) are led by teams of senior undergraduate Computer Science (CS) and Computer Systems Engineering (CSE) students.

Both groups participated in the creation of learning and gaming systems, and individual teams were composed of three to five members. Additionally, source code from a tutoring system (NSF supported project) was created jointly by a team of undergraduate and graduate student. The majority of participants were undergraduate students because a special interest exists in measuring the usefulness of the adoption of the proposed approach in communities (such as HCI and Educational Technology) where developer experience or developing resources are limited. Senior undergraduate students are by no means experts yet, even though exceptional cases can exist. The ASU CS curriculum describes Capstone project I (CSE 485) and Capstone project II (CSE 486) as courses that help students to emphasize development process and technical skills, along with teamwork and communication.

Teams defined their own functional requirements, usability requirements related to affect measurement, and adaptation logics. Teams were asked to develop their projects iteratively, covering analysis, design, implementation, and testing in each iteration. They were also asked to deliver a working product that was assessed based on its quality and the user experience it offered. It was established that the assessment criteria would focus on the quality of the resulting product irrespective of the computational complexity achieved, i.e., they were asked to deliver a user experience to the best of their ability.

All teams were provided with access to hardware and software on a computer server, Intel i7 @ 2.4 GHz processors with 4 GB of RAM running MS Windows 7 (later updated to 8 and 10), with all sensor devices installed and running. However, several teams set up the sensors and required software on their own computers too.

7.2. Projects

Over seven years, 11 projects were developed as depicted below (Figure 7.1) and described in the following two sections; 3 were professional research projects and 8 were undergraduate capstone projects. The engineering effort for the capstone projects is presented here:

a) an academic semester (4 months) was allotted to defining requirements, designing and developing the functional logic; and

b) a second academic semester (4 months) was used for developing and testing.

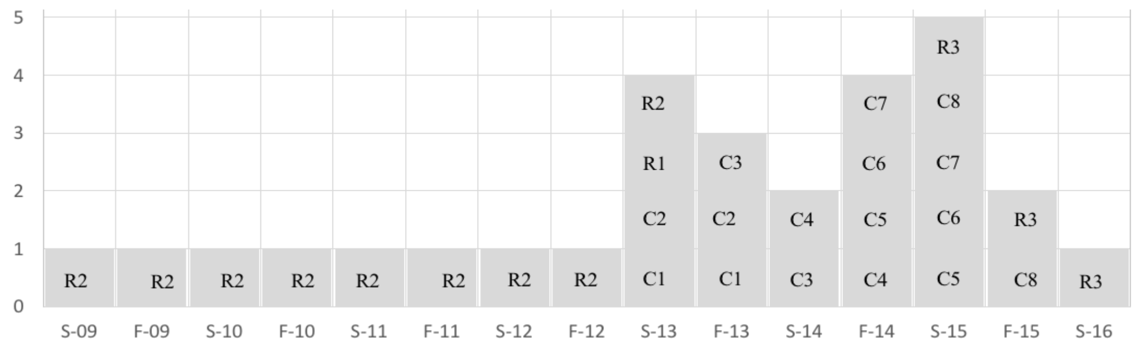


Figure 7.1. Case study development from Spring 2009 to Spring 2016. Columns represent semesters and boxes enclose the ID of the projects under development during that semester.

Capstone teams were formed of 4 to 5 students who dedicated around 5 hours per week each, providing an approximate total of 320 to 400 hours per team per semester.

Undergraduate and graduate students participating in research projects were hired for 20 hours per week for the time the project was running.

7.3. Projects by Research Staff

Research projects that were analyzed as case studies include:

a) Closed-loop system to analyze emotional impact on human interaction in virtual world environments; a project led by graduate research assistants in the ANGLE Lab with ONR support.
b) Affective Meta Tutor (AMT) source code, supported by NSF.
c) Enhanced affective support in AMT; a doctoral project.

These are described in the following subsections.

7.3.1. Project R1: Affect in Virtual Worlds

Period: Spring 2013

Team: 3 graduate students from the ANGLE Lab staff.

Description: This project focused on the role of avatars in interpersonal communication. The project integrated a generic, real-time, multimodal affect recognition hub as an input in an online virtual world to make an avatar mirror its user's affective state. Affect vectors (determined by PAD coordinates) in a continuous affective space were applied to characterize the user's affective state in real time.

Details: This system introduced affect into an online virtual world. The system was developed to analyze the importance of affect in relation to communication between users in virtual worlds and to explore approaches for improving virtual interactions. The system was developed using the Java platform. The system focused on performance, specifically the ability to achieve a real-time loop showing an animated representation of the user's current affective state. To simplify, PAD vectors were discretized into four categories: frustrated (low pleasure, high arousal, and low dominance), engaged (high pleasure, high arousal, and high dominance), bored (low pleasure, low arousal, and low dominance), and meditating (high pleasure, low arousal, and high dominance).

Structure: The system is composed of three parts as shown in Figure 3.7 in Chapter 3. It comprises the following components:

a) A virtual world, a commercial off-the-shelf system named Second Life.
b) A functional logic that acts as an interface between the virtual world and the ADAS affect recognition logic. The interface triggers native system input events (equivalent to keyboard strokes) that change the avatar according to an affect vector value.
c) the ADAS affect recognition logic using one agent, i.e., the core components sensing and perception.

Results: Feasibility of embedded affect demonstration in avatar behaviors was shown to be viable mirroring user affective states of frustration, engagement, boredom, and concentration, as well as neutrality, using an online virtual world avatar. Efforts to improve human-human communication in virtual environments were inconclusive, diverse issues were explored. The technical feasibility was shown at a demonstration presented at ACII 2013 (Gonzalez-Sanchez et al. 2013).

7.3.2. Project R2: Affective Meta Tutor

Period: Spring 2009 - Fall 2012

Team: Several undergraduate students coordinated by a team of graduate students from the NSF project.

Description: The AMT system comprised (1) a tutor that taught system dynamics modeling, (2) a meta-tutor that taught good strategies for learning from the tutor, and (3) an affective learning companion that encouraged students to use the learning strategy that the meta-tutor recommended. The affective learning companion’s messages were selected on the basis of the user’s affective states. Affective interventions are realized between tasks, when the student completes one task and is starting the next. The results were evaluated by comparing the learning gains under three conditions: with the tutor alone; with the tutor and the meta-tutor; and with the tutor, the meta-tutor, and the affective learning companion.

Structure: The system comprises the following components:

a) an affective companion, which uses legacy code for affect recognition;
b) a meta-tutor, which uses a production rule engine (Drools) to handle adaptation rules; and
c) an intelligent tutoring system.

Results: Multiple experiments were run with this tool from 2009 to 2013 and, based on the results relating to affective learning companions, it is hypothesized that these interactions works best with simple short tasks, perhaps because there are more frequent opportunities for interventions between tasks. Improving affective learning companions by presenting interventions at affectively appropriate times is mentioned as part of the authors’ future work (VanLehn et al. 2014; Zhang et al. 2014).

7.3.3. Project R3: Enhanced Affective Tutor

Period: Spring 2015 - Spring 2016

Team: 3 staff and volunteers from the ANGLE Lab staff.

Description: As an evolution of project R2, this new system eliminated the meta-tutoring component and focused on the affective support. Taking the R2 software components for the tutor and meta-tutor, a new system was developed in which the affect recognition and affect-driven reactions were reengineered. The agent interventions are continuous and not limited to pauses between task.

Details: A 2D character was used to assist the user to achieve goals in a tutoring environment. It provided direct instructions, practical support, and guidance, while showing empathetic behavior. This system was developed using the Java platform. Measurements of engagement, boredom, and frustration, as well as information about user's interactions with the tutor interface were used to adapt the interventions made by the companion (content and timing) to improve the learning process.

Structure: The system comprises the following components:

a) an affective companion, which uses ADAS affect recognition and adaptation rules; and
b) an intelligent tutoring system.

Results: This project is being used as a tool to support a doctoral dissertation under development at ASU. It is a tutoring system that extends the work done in R2, implementing and evaluating an adaptive affective agent focused on discouraging boredom, promoting engagement, and inducing or maintaining a productive learning path.

7.4. Projects by Capstone Students

Every senior undergraduate student in the CS and CSE programs at ASU is required to take a two-semester capstone course. This gives students an opportunity to apply their technical skills and knowledge of engineering principles and software development within a complex, team-oriented software project, system, or device. Students must work as a team over two semesters towards a fairly well-defined goal. The team meets and communicates with a designated representative (mentor) at regular intervals throughout the course of the project. The students are expected to gather requirements for a problem and then design, develop, test, and deploy a solution. The mentor does not act as a project manager.

Under Dr. Robert Atkinson and ANGLE Lab sponsorship, I submitted 12 proposals for ASU CSE capstone projects between the Spring 2013 and the Spring 2015 semesters. It was expected that participating students would have the following skills:

a) a software development background;

b) a proficiency in a programming language; and

c) an interest in human-computer interaction and/or user-interface design.

The latter was not essential but was deemed to be an advantage.

I participated in these projects as a mentor. For each proposed project we, as a lab, provided teams with mentoring and the tools (hardware and software) that support emotion and interest recognition, while challenging them to imagine, design, and build a system that exploits those capabilities, including, but not limited to, a next-generation learning environment, a more engaging video-game, or an empathetic health support system. We encouraged and supported students to present their work at international conferences as a demo, poster, or short paper. Regular meetings were held to provide feedback and support, but project management was the sole responsibility of students.

From the 12 proposals presented, 8 were selected by students and worked on by a designated team for a two-semester period. The following subsections summarize the proposals presented. Appendix B presents the capstone faculty project proposal template submitted to the ASU capstone managers. Submitted proposals follow the template and each includes slight variations.

Grades are those reported by the mentor or mentors and they do not necessarily correspond to the final evaluations of the students by the professor responsible for the course overall. Grades are based not only on product development, but also on aspects such as teamwork, written communication, meeting outcomes, planning and project management, documentation, use of resources, and final deliverables.

7.4.1. Project C1: Affect Recognition

Period: Spring 2013 - Fall 2013

Team: Capstone team with 3 undergraduate students.

Description: The goal of this project was to inspire and support student creativity in the production of an affective adaptive environment of original design. We provided the devices and expertise, and the students were encouraged to create an interactive environment that took advantage of multimodal affective adaptation capabilities. By the end of the year the students were expected to deliver an interactive computational environment of their own choosing (such as a learning environment or a videogame) that was able to self-adapt in response to a user’s performance and affective states.

Details: The goal of this project was to measure the difficulties in building affect-driven adaptive systems without the support of ADASPL. Thus, teams were provided with devices, but not the ADAS software, and instructed to create an affective adaptive system with them. They were provided with the hardware's proprietary SDKs and third-party libraries. We required them to achieve a relatively simple task: collect data from multiple sensors (at least one that uses a serial port and one that uses a network), integrate the information (synchronize and merge), and then share it with either the client system of their choice or one of their own development.

Results: Students were able to produce a beta system for multimodal affect recognition but did not construct a system that used that data. The final result was a system that gathered information, synchronized it, and transformed it into PAD values. It read from a serial port sensor, a skin conductance bracelet, and the Emotiv EPOC headset. It was also able to take log files of previously collected data as an input for simulation purposes.

Student grade: B-

Team report: Students reported difficulties regarding:

- dealing with bugs in the third-party libraries.
- dealing with hardware-software connection validations.
- dealing with the implementation of the task described in the sensing, perception, integration, and synapse components, including dealing with numerous data formats and synchronizing data.
- scheduling and intra-team conflicts.

7.4.2. Project C2: Affective Companion with a 2D Avatar

Period: Spring 2013 - Fall 2013

Team: Capstone team with 4 undergraduate students.

Description: This project required gathering affect information using physiological sensors and creating an intelligent virtual character able to show a high level of understanding of the meaning of the user's changes in affective state. The affective companion was intended to function inside a learning environment; therefore, the system focused on achieving a social relationship with students and helping them to improve their performance.

Details: This team was provided with hardware and a beta version of the ADAS framework. The team developed a 2D dynamic image (character), an affective companion, which was integrated into a computer-based learning environment (already developed and focused on teaching the modeling of dynamic systems). The character was aware of user actions and affective states and reacted by displaying messages and changing its expressions.

Results: The team focused their work on creating an avatar, including pictures, gestures, and design; they also made minor advances in making a connection with affect recognition.

Student grade: A-

Team report: Students reported difficulties because:

- the group did not meet at regular intervals.
- they mainly worked on what they described as a distributed development environment.

7.4.3. Project C3: Affective Pac-Man Videogame

Period: Fall 2013 - Spring 2014

Team: Capstone team with 5 undergraduate students.

Description: The goal of this project was to inspire and support student creativity in the production of an affective adaptive environment of original design. We provided the tools, software components, and expertise, while students were encouraged to create an iterative environment that took advantage of multimodal affective adaptation capabilities. By the end of the year the students were expected to deliver an interactive computational environment (such as a learning environment or a videogame) that was able to self-adapt in response to a user's performance and affective states.

Details: This team was provided with both hardware and the ADAS framework. Affect-driven adaptive capabilities were incorporated into an open version of the well-known video game, Pac-Man. The project was developed using the Java platform. Affect recognition was used to drive changes in the game, aiming to improve the user experience by maintaining or increasing the player's engagement. The strategy for keeping the user engaged was defined as follows: (1) boredom was undesirable, and if the player approached this state the game was made far more difficult by decreasing the speed of Pac-Man and increasing the number and speed of the ghosts as well as the music tempo; (2) frustration was undesirable, and if the user reached this state the game was made easier by increasing the speed of Pac-Man, decreasing the number and speed of the ghosts and the music tempo, and also providing additional special features such as power pellets and fruits; and finally (3) a state of meditation was undesirable, and if the user approached this state the game was made slightly more difficult by increasing the speed of Pac-Man and the music tempo in order to push the player toward a state of engagement. The system concerns in this case involved both performance and improvement in the user experience over time. The project evaluated the ADAS affect recognition logic and its interfaces with an adaptation logic.

Structure: The C3 system was composed of three parts as shown in Figure 3.9 in Chapter 3 and described here:

a) An already implemented functional logic. An open source implementation of the Pac-Man video game was used. It was modified to include effector points and an adaptation logic.
b) Effectors added to the functional logic.
c) An adaptation logic and a reaction engine implemented and connected with ADAS affect recognition logic. Adaptation logic implementation was not restricted to following the ADASPL approach.

Results: The team successfully accomplished the goal of altering the Pac-Man game in order to elicit emotional responses from users. An extended abstract was published and a demo presented at ISWC 2014.

Student grade: A

Team report: Students reported the successful implementation of the game and highlighted their learning outcomes.

7.4.4. Project C4: Affective Horror Game

Period: Spring 2014 - Fall 2014

Team: Capstone team with 5 undergraduate students.

Description: This project aimed to leverage the inclusion of affect-driven adaptation capabilities in gaming environments by inspiring and supporting student creativity in the development of an affective adaptive game or gaming environment of original design. By the end of the year it was expected that the students would deliver an interactive computational game or gaming environment that was capable of affect-driven adaptation.

Details: This team was provided with both hardware and the ADAS framework. The game was puzzle-oriented with horror-type content. It contained stressful and scary scenes, including hostile entities, monsters pursuing the protagonist, etc. It focused on the frightened and calm emotional states. The game aimed to use emotional information, specifically the user's fear, and the user's navigational decisions as the basis for changes in the game environment. The game environment was developed using Unreal Engine 4 and a back-end coded in C++.

Structure: The C4 system was composed of three parts as shown in Figure 3.8 in Chapter 3 and described here:

a) A functional logic created in a game engine. The functional logic was implemented from scratch in Unreal Engine 4.
b) Effectors added to the functional logic.
c) An adaptation logic and reaction engine implemented and connected with the affect recognition logic. The adaptation logic was implemented in Java but was not restricted to following the ADAS approach. The project evaluated the ADAS affect recognition logic and its interfaces with an adaptation logic.

Results: The project was completed but the team was more interested in the design of the graphical elements of the game than in its functionality or its adaptive or affective capabilities.

Student grade: A

Team report: Students reported difficulties regarding:

- the time spent learning how to use first the Unity game engine and then Unreal Engine 4.
- modeling the affective state of fear from the constructs provided by the used sensor devices.

7.4.5. Project C5: Affective Working-Memory Game

Period: Fall 2014 - Spring 2015

Team: Capstone team with 6 undergraduate students.

Description: This system aimed to leverage the incorporation of affect-driven adaptation capabilities into a game environment by inspiring and supporting student creativity in the development of an affective adaptive system of original design. By the end of the year, it was expected that the students would deliver an affect-driven self-adaptive environment. It was expected that the students would include assets such as software design, testing, validation, and documentation.

Details: This team was provided with both hardware and the ADAS framework. The project aimed to find a correlation between affective states and short-term spatial memory retention. The game environment was developed using C# and emulating a previous version available in MATLAB (Lewandowsky, Oberauer, Yang & Ecker 2010). The implemented game is an adaptation of the spatial short-term memory level of this game. The game is a series of memory tests using dots on a grid. In a given level, the system selects a number of locations on the grid to place dots. These dots are temporarily and simultaneously revealed to the user; when the dots disappear, the user has to remember and select the positions in which the dots had appeared.

The team built and tested a system that changed a game according to the affective state and performance of the user. In particular, the user's frustration levels and performance score were used as the basis for adaptations in the level of difficulty. The game scored the user's attempt using a distance-scoring algorithm. Every five levels, the system determined whether to increase or decrease the size of the grid and the number of dots in the grid based on the user's distance scores and the PAD values for the previous five levels. The game consisted of five sets of five levels.

Structure: The C5 system was composed of three parts, as shown in Figure 3.8 in Chapter 3 and described here:

a) A functional logic created in a game engine. The functional logic was implemented from scratch in C#.

b) Effectors and probes added to the functional logic.

c) An adaptation logic and reaction engine implemented and connected with the affect recognition logic and functional logic. The adaptation logic was implemented in C#.

Results: A short paper was submitted to FDG 2015, but it was not accepted for publication.

Student grade: A

Team report: Students reported:

- a high CPU demand for ADAS when running continuously for long time, suggesting a possible memory leak.

7.4.6. Project C6: Affective Shooting Game

Period: Fall 2014 - Spring 2015

Team: Capstone team with 5 undergraduate students.

Description: The project aimed to leverage the inclusion of affect-driven adaptation capabilities in either existing or new video games. In particular, this project focused in particular on amusing and highly-interactive video games that aimed not only to provide entertainment but also to support learning, health improvement, and/or any other social or personal outcomes. By the end of the year it was expected that the students would deliver an environment capable of affect-driven adaptation. It was expected that the students would include assets such as software design, testing, validation, and documentation.

Details: Students were inspired by project C4 and aimed to emulate the results by increasing the number of affective states involved.

Structure: The structure for project C6 project was comparable to that for project C4.

Results: The project was completed but the team was more interested in the design of the game's graphical elements than its functionality.

Student grade: A

Team report: Team reported difficulties regarding:

- the time spent learning how to use both the Unity game engine and Unreal Engine 4.
- hardware requirements for executing the game as they had planned.

<u>7.4.7. Project C7: Affective Companion with a 3D Avatar</u>

Period: Fall 2014 - Spring 2015

Team: Capstone team with 4 undergraduate students.

Description: We aimed to extend the functionality of virtual companions by adding empathy (affect-driven adaptation) as a capability. By the end of the year it was expected that the team would: (a) deliver a virtual companion able to show empathetic behaviors, achieved by exploring diverse options for its design and (b) integrate the virtual companion into an already developed (and provided) learning environment.

Details: This team was provided with hardware and the ADAS framework. The team developed a 3D dynamic image (character), an affective companion. They did not accomplish its integration into a computer-based learning environment. The character was aware of user actions and affective states and reacted by displaying body gestures.

Results: The team focused their work on creating a 3D character using motion capture, a tools to create 3D human avatars[4], and an open source game engine[5] in the Java platform. They made minor advances in making a connection with affect recognition. They advanced from a 2D cartoon to a realistic 3D human figure, and they placed it in a 3D game engine. Motion capture technology was used to record and apply realistic animation sequences to the character, including: walking, waiting, head shaking, head nodding, shrugging, head scratching, waving, and crossing arms.

Student grade: A

[4] www.makehuman.org

[5] www.jmonkeyengine.org

Team report: Students reported:

- as the team members had diverse interests, they each acted on an individual basis.

7.4.8. Project C8: Affect Recognition in Video Clips

Period: Spring 2015 - Fall 2015

Team: Capstone team with 4 undergraduate students.

Description: We sought to validate the PAD scheme as suitable for use in representing multimodal affect information. The team was provided with sensors and the ADAS framework. The stimuli used for this project were photos and videos pre-selected to elicit very specific affective reactions. By the end of the year it was expected that students would complete the system as well as design and implement the study. The students worked in collaboration with the research team mentoring this project on the data analysis and the interpretation of the results.

Details: Project was incomplete. Its final stage was limited to a real-time comparison of the emotion expected and emotion inferred from a user watching a specific video. It was used pre-classified video clips selected from a databases named DEAP (Koelstra et al. 2012). It included for each video a self-report and sensor data.

Structure: The structure for project C8 was comparable to that for project C5.

Results: The first semester of work was very productive, but progress declined in the second semester.

Student grade: B

Team report: Team reported:

- communication issues.

7.4.9. Additional Projects Proposed but Not Realized

Four other projects were proposed but not selected by students in the capstone course. They are listed below just to show additional potential diversity. The outlines for these projects were as follows:

a) Affective Companions for Learning Environments (Fall 2013 - Spring 2014). This project requires gathering affect information using physiological sensors and creating an intelligent virtual character able to show a high level of understanding of the significance of changes in the user's behavior. The affective companion is envisioned as working inside a learning environment; therefore, it focuses on achieving a social relationship with students and helping them to improve their performance. Previous work has been done in this area and so the team will work with an existing version of this affective companion.

b) Empathetic Virtual Companions for Learning Environments (Spring 2014 - Fall 2014). This project aims to extend the functionality of virtual companions by adding the capability for empathy. This project is the second phase of a previously completed project. The goals to be accomplished include developing or completing an implementation that allows students to: (1) move through a set of diverse tasks to master a particular concept or skill; (2) solve each task step-by-step; (3) receive feedback and hints; and (4) receive affective support.

c) Exploring the Inclusion of affect-driven adaptive capabilities in apps for the Google Glass platform or Apple mobile devices (Spring 2014 - Spring 2015). This aims to leverage the inclusion of affect-driven adaptation capabilities in mobile devices through inspiring and supporting student creativity in the development of an affective adaptive app of original design for either the Google Glass or Apple mobile devices. By the end of the year it was expected that the students deliver an interactive app able to adapt itself on the basis of the user's affective states.

d) Learning Environment and Companion with Affective Adaptive Capabilities (Spring 2014 - Spring 2015). This project aims to leverage the inclusion of affect-driven adaptation capabilities in learning environments by inspiring and supporting student creativity in the development of an affective adaptive learning environment of original design. By the end of the year it was expected that the students deliver an interactive computational learning environment capable of affect-driven self-adaptation.

7.5. Projects Selected for Analysis

The broad range of projects supports the claim to the proposed approach's generality, which is discussed later. However, for a complete evaluation of the approach, it is important to measure the properties of the products and the engineering process applied for the approach evaluation. To do so, from the 11 systems developed, I selected a subset of 7 systems with varying complexity, ranging from those with basic features to those integrating diverse components. This selection discards two unfinished projects. For each of the projects not included in the selection there is another project that accomplished similar objectives, allowing for the appropriate assessment of all objectives offered by the ADASPL approach:

a) C2 and C7 are represented by R1

b) C6 was not completed and is represented by C4, which has similar characteristics.

c) C8 is represented by C3.

The selected systems are summarized in Table 7.3 and described below:

a) Project C1. An unguided implementation of a functionality similar to the ADAS affect recognition logic.

b) Project R1. A system with low-level complexity using the ADAS affect recognition logic's core components (sensing and perception) without customization or extension. It is a reactive system, i.e., no adaptation mechanisms were implemented.

c) Project C3 (Pac-Man). A system with low-level complexity using the ADAS affect recognition logic's core components (sensing, perception, integration and synapse) and an adaptation logic that customizes the component engine (without an introspection component). It is an adaptive system.

d) Project C4 (Horror Game). A system with low-level complexity using the ADAS affect recognition logic's core components (sensing, perception, integration and synapse) and an adaptation logic that customizes the component engine (without an introspection component). It is an adaptive system.

e) Project C5 (Memory Game). A system with low-level complexity using the ADAS affect recognition logic's core components (sensing, perception, integration and synapse) and an adaptation logic with the component engine customized to apply condition-action pairs (with an introspection component).

f) Project R2 (AMT). A tutor implementing a functionality similar to the ADAS affect recognition and adaptation logics without using ADAS at all.

g) Project R3 (Enhanced Tutor). A tutor using the ADAS affect recognition logic's core components (sensing, perception, integration and synapse) and an adaptation logic with the component engine customized to apply a production-rule system (with an introspection component).

Additionally, the following comparisons were also made:

a) C1 is compared with the ADAS affect recognition logic.

b) R2 (AMT) is compared with R3 (Enhanced Tutor).

Table 7.3. Summary of Case Studies Selected for Further Analysis

ID	Deployment scenario	ADAS AR	ADAS adaptation	Probe (introspection)	Effector
C1					
R1	Scenario A	Monomodal		☐	☐
C3	Scenario C	Multimodal	Conditional	☐	☑
C4	Scenario B	Multimodal		☐	☐
R2					
R3	Scenario C	Multimodal	Rule engine	☑	☑

The project process measures and software metrics for these projects are described in the next chapter.

7.6. Summary

In this chapter, the case studies conducted to evaluate the approach presented in this dissertation were introduced. They exemplify diverse architectural configurations, uses, and customizations of the infrastructure and tools, as well as the application of the guidelines for the engineering process. Chapter 8 provides measures and metrics related to each project. These measures and metrics characterize the engineering process and the products created in diverse scenarios and allow for their comparison.

CHAPTER 8
ANALYSIS OF PROCESS AND PRODUCTS

Three kinds of metrics were collected for each project to quantify the development process and the properties of the software product, as described in section 2.4: (1) process measures that quantify the involvement of people in the project over time; (2) software complexity metrics that quantify the complexity of the software; and (3) software structure metrics related to design flaws that quantify the structural complexity of the software. These measures and metrics are presented for projects C1, R1, C3, C4, R2, and R3. Reports for code complexity metrics generated by RSM are compiled in Appendix C.

8.1. Analysis of Project C1

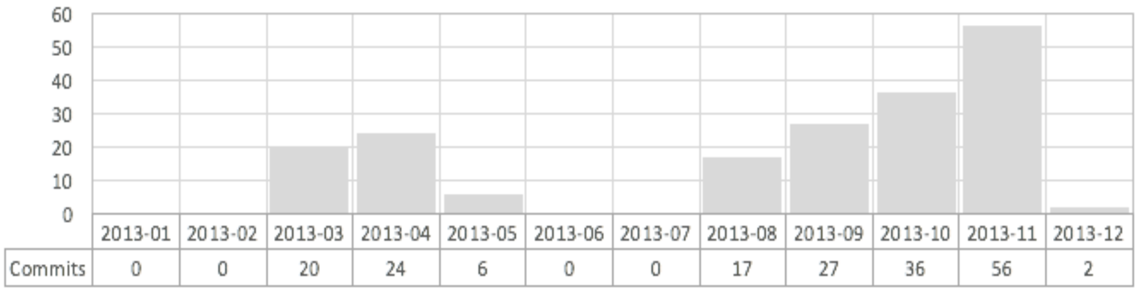
Project C1 built an affect recognition logic from scratch. This project is compared with the ADAS affect recognition infrastructure.

8.1.1. Process Measures

This was a two-semester project running from Spring 2013 to Fall 2013. Commits by date as registered in the Git repository are summarized in Figure 8.1. From the number of commits, it is apparent that the coding activity begins in March 2013, stops during the summer, and increases exponentially towards the end of the school year.

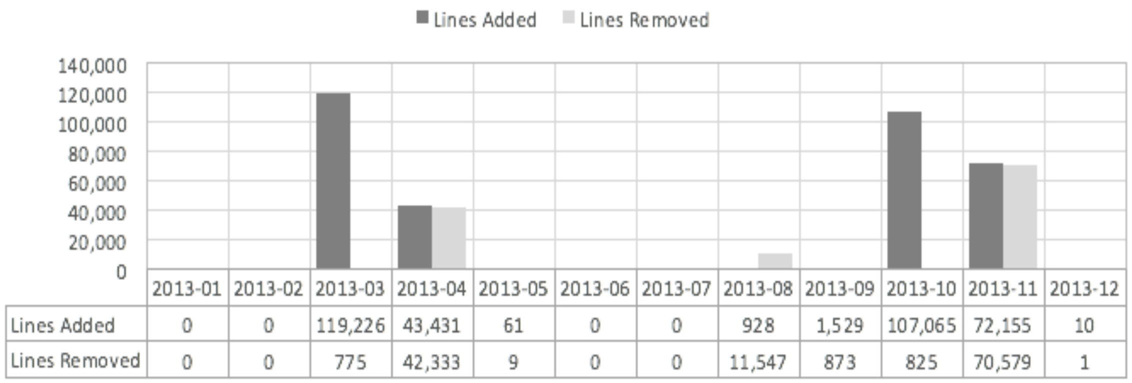
Each commit adds or removes lines in the repository. Figure 8.2 shows the number of lines added and removed, grouped by month.

The repository finished with 217,463 lines (344,405 added, 126,942 removed) in 114 files, but only 61 files and 2755 lines correspond to the source code files. The historical growth of the number of lines stored in the repository is depicted in Figure 8.3.



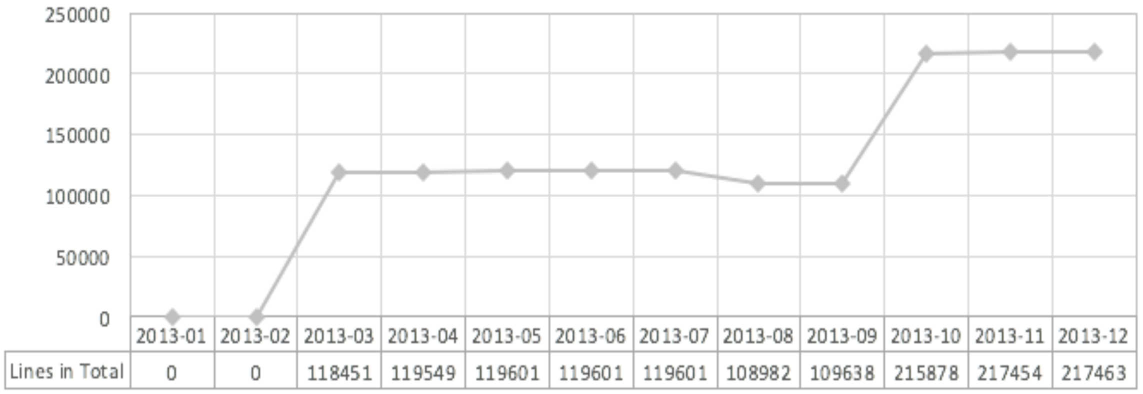
	2013-01	2013-02	2013-03	2013-04	2013-05	2013-06	2013-07	2013-08	2013-09	2013-10	2013-11	2013-12
Commits	0	0	20	24	6	0	0	17	27	36	56	2

Figure 8.1. Commits per month in project C1.



	2013-01	2013-02	2013-03	2013-04	2013-05	2013-06	2013-07	2013-08	2013-09	2013-10	2013-11	2013-12
Lines Added	0	0	119,226	43,431	61	0	0	928	1,529	107,065	72,155	10
Lines Removed	0	0	775	42,333	9	0	0	11,547	873	825	70,579	1

Figure 8.2. Lines added and removed per month in project C1.



	2013-01	2013-02	2013-03	2013-04	2013-05	2013-06	2013-07	2013-08	2013-09	2013-10	2013-11	2013-12
Lines in Total	0	0	118451	119549	119601	119601	119601	108982	109638	215878	217454	217463

Figure 8.3. Number of lines per month in project C1.

The project was developed by a 3-member team. However, the contributions submitted to the repository suggest that most of the code can be attributed to one author. Table 8.1 summarizes the information. Authors' names correspond to the usernames that students used in the Git repository. 'Mentor' is a wildcard used to represent all of the mentoring staff, not a specific person, and the data for the 'Mentor' row corresponds to the sum of the activities of all the users in the mentoring staff.

The system has 2,209 LOC in 53 files. Table 8.2 summarizes the development effort.

Table 8.1. Contributions per Author in Project C1

Author	Commits (%)	Lines added	Lines removed
Austin	64.36%	335,869	131,786
Tom	21.81%	930	110
Matt	10.63%	1,060	163
Mentor	3.19%	11,643	5

Table 8.2. Effort Estimation for Project C1

Project	Staffing (person)	Schedule (month)	LOC	Ratio
C1	3	7	2,209	105.19

8.1.2. Software Metrics

Table 8.3 shows: (1) a row with the software complexity metrics for project C1; a row with metrics for the ADAS affect recognition logic (ADAS AR) which includes the packages `adas.ar`, `adas.gui`, and `adas.toolkit`; and (3) a comparison of the two made in the final row, where differences are indicated as improved (+), deteriorated (−), or the same (0). This will be discussed further later, but for now it is useful to show these side by side for easy reference. The metrics for each of the ADAS packages are presented in section 8.7.

Table 8.4 shows the software structural metrics for the project. The structural metrics are calculated for each package. The most significant numbers for project C1 correspond to the largest package shown in Figure 8.4, `myStuff`, which is the core of the project. Notice that it occupies a significantly complex part of the dependency graph. Figures 8.4 and 8.5 depict the composition of project C1 and ADAS AR, respectively. A graphical look at the structure of the project components and the ADAS AR provide an overview of the extension of both systems. It is worth remarking that project C1 was not fully finished and is therefore an incomplete implementation compared to the ADAS AR.

Table 8.3. Complexity Metrics for Project C1 and for the ADAS Affect Recognition Logic

Package	**Files**	**LOC**	**DIT**	**NOC**	**Methods**	**CC**		
						Avg.	**Max.**	**Min.**
C1	53	2,209	1.00	0.07	243	1.58	24	1
ADAS-AR	61	2916	1.23	0.19	318	1.96	46	1
Difference	0	+	+	+	+	+	+	0

Table 8.4. Structural Metrics for Project C1

Package	**FAT**	**H**	**Ca**	**Ce**	**I**	**A**	**|D|**
deo	0	0.5	3	1	**0.25**	**0**	0.75
data	9	1.42	0	0	1	0.29	0.29
myStuff	69	2.18	3	1	0.25	0.03	0.72
classSerialization	3	1	0	0	1	0.25	0.25
scriptEngine	1	1	1	1	0.5	0	0.50
view	3	0.66	0	1	1	0	0

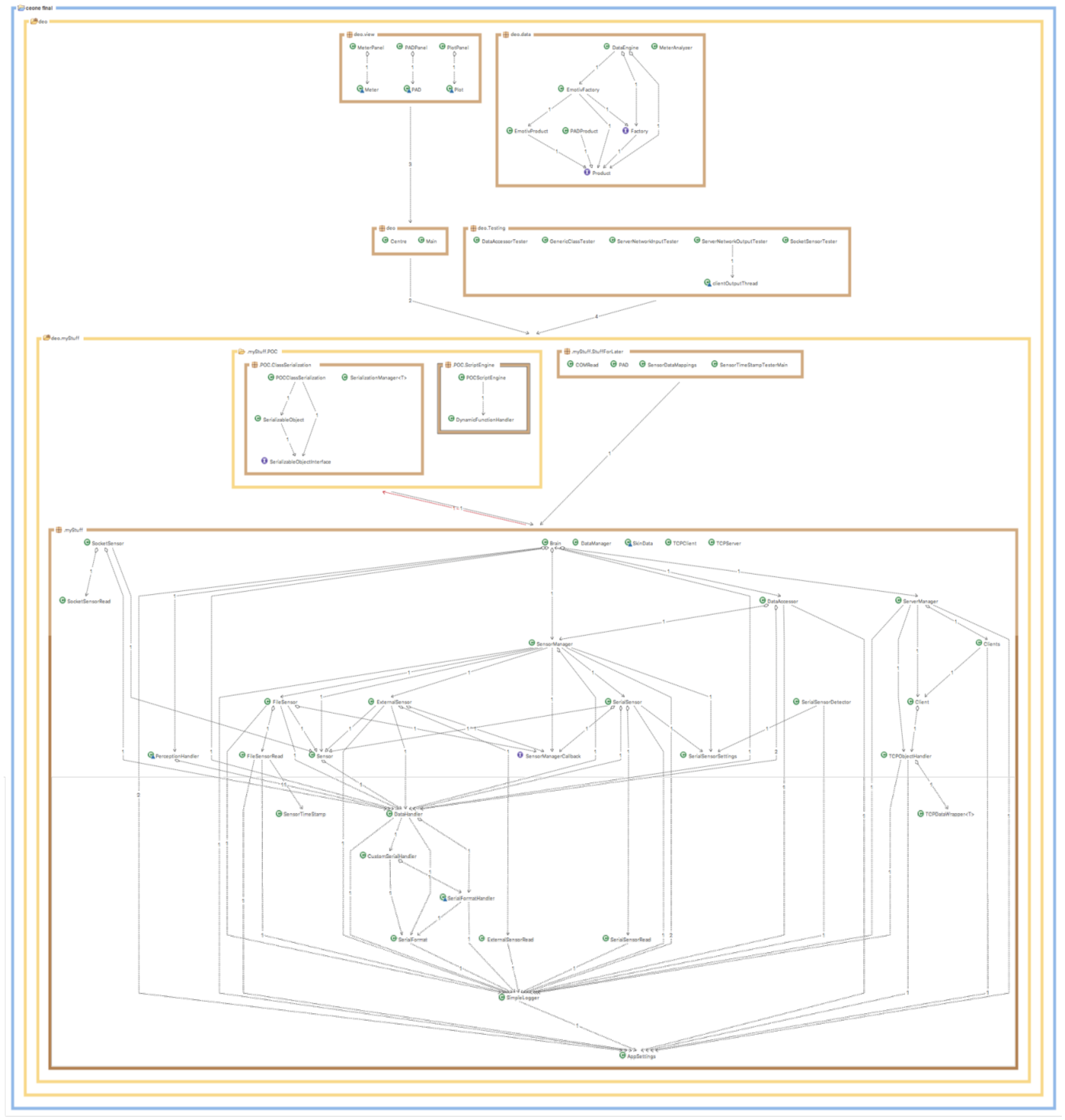


Figure 8.4. Dependency graph depicting the composition of project C1.

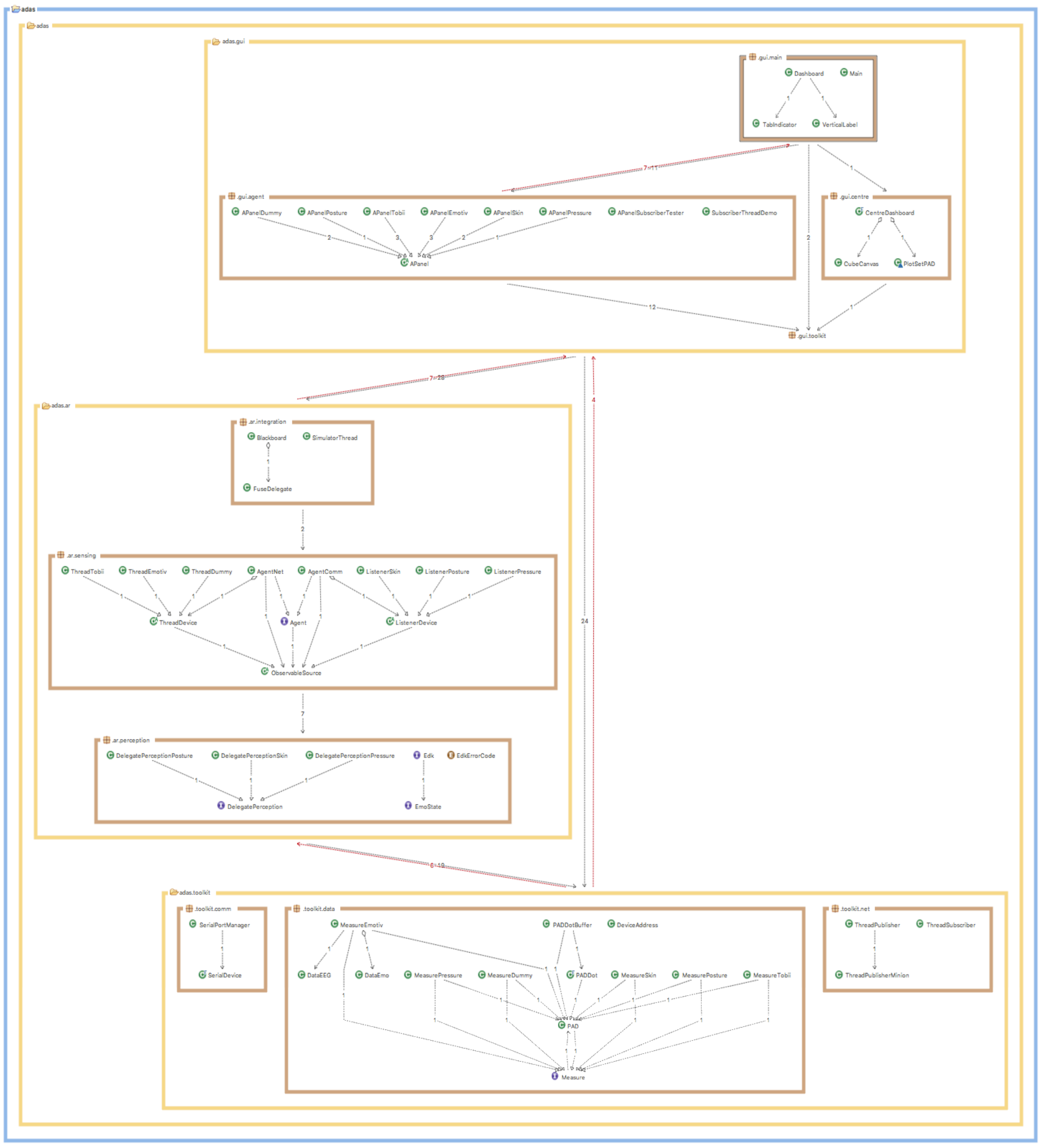


Figure 8.5. Dependency graph depicting the composition of the ADAS AR.

8.2. Analysis of Project R1

Project R1 built a system that corresponds to scenario A, as described in section 3.5.1, a monomodal affect recognition system for affect awareness, i.e., a system without adaptation capabilities. Project R1 demonstrates the feasibility of implementation for scenario A.

8.2.1. Process Measurements

This was a one-semester project running in Spring 2013. As this project was developed by 3 staff members, and had the goal of showing feasibility in the simplest scenario, no measures were registered for the process, neither commits per date and author, nor lines added or removed per date and author. However, this group reported 4 weeks of developing work.

The system has 181 LOC in one file. Even though 3 staff members participated, one did most of the development and the others ran testing and performed studies related to emotional expression in virtual worlds, as described in Gonzalez-Sanchez et al. (2013). Table 8.5 summarizes effort estimation measures that give a ratio of 181.0 LOC per person per month.

Table 8.5. Effort Estimation for Project R1

Project	Staffing (person)	Schedule (month)	LOC	Ratio
R1	1	1	181	181.0

8.2.2. Software Metrics

The R1 functional logic consists of a single class. The software complexity metrics and software structural metrics for project R1 are shown in Tables 8.6 and 8.7, respectively. The code complexity metrics show an average complexity for R1. As it is a single-file project structural metrics do not apply because cohesion and coupling cannot be assessed with a single file, i.e., with a dependency graph with only one node.

Table 8.6. Complexity Metrics for Project R1

Package	Files	LOC	DIT	NOC	Methods	CC		
						Avg.	Max.	Min.
R1	1	181	1	0	5	8.80	17	1

Table 8.7. Structural Metrics for Project R1

Package	FAT	H	Ca	Ce	I	A	|D|
R1	0	1	0	0	**1**	**0**	**0**

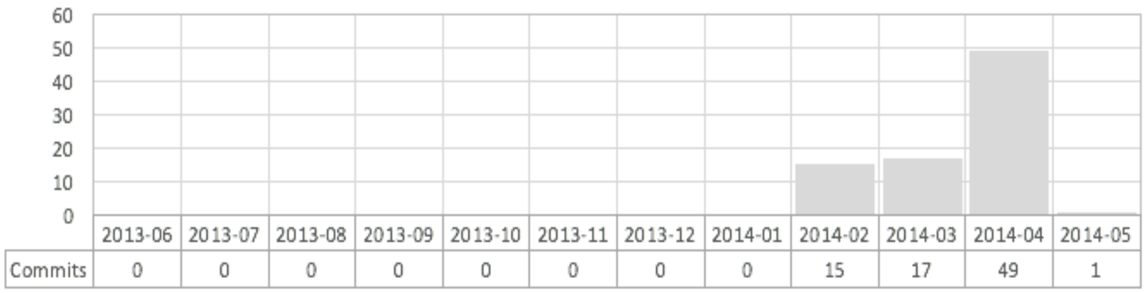
8.3. Analysis of Project C3

Project C3, the Pac-Man video game, built a system that corresponds to scenario C, as described in section 3.5.3, a multimodal affect recognition and adaptive system. Project C3 shows the feasibility of modifying a third-party system to include affect-driven capabilities.

8.3.1. Process Measurements

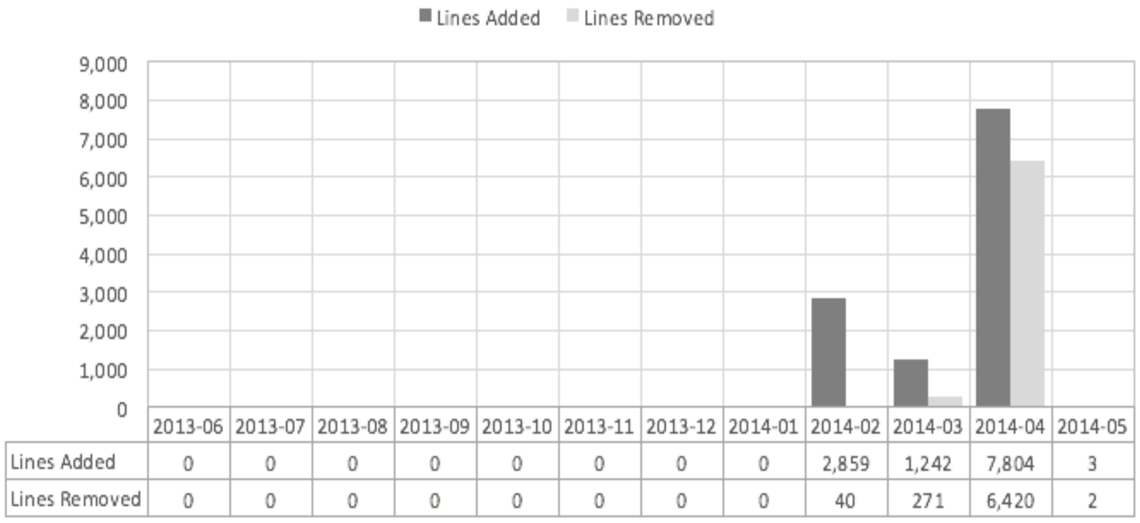
This was a two-semester project running in Fall 2013 and Spring 2014. Figure 8.6 summarizes the commits by date as registered in the Git repository. It can be seen that the coding was in fact realized in 4 months (in the second semester of the capstone course), during Spring 2014. Each commit added or removed lines in the repository. Figure 8.7 shows the amount of lines added and removed per month. The repository started with approximately 20 files and 3,000 lines; 11 of those files were Java files. These increased steadily to 44 files and a total of 5175 lines, of which 14 were Java files with 2,566 LOC. Figure 8.8 shows the total amount of lines per month.

It is evident that of the 5 students involved in the process, 2 were more active than the others in the percentage of commits made. Table 8.8 summarizes the information. Authors' names correspond to the usernames that students used to register in the Git repository. As in the previous project, 'Mentor' is a wildcard term for the mentoring staff.



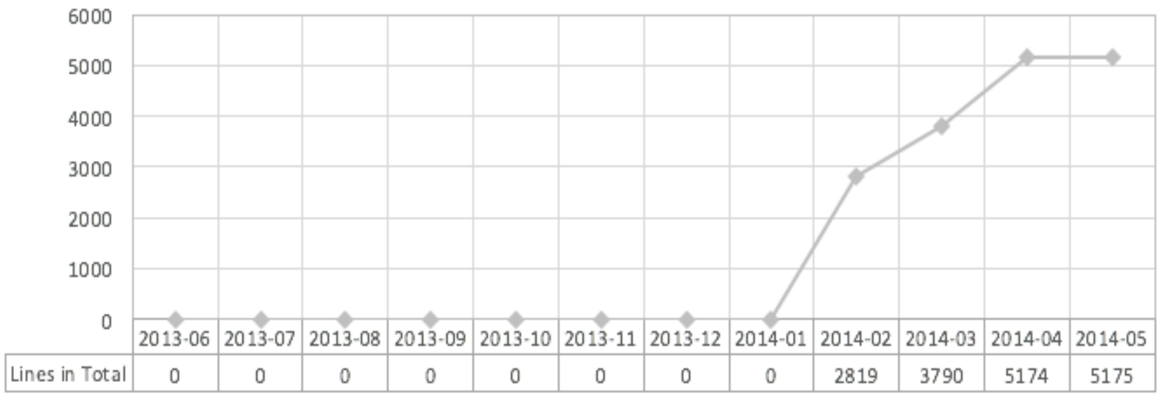
	2013-06	2013-07	2013-08	2013-09	2013-10	2013-11	2013-12	2014-01	2014-02	2014-03	2014-04	2014-05
Commits	0	0	0	0	0	0	0	0	15	17	49	1

Figure 8.6. Commits per month in project C3.



	2013-06	2013-07	2013-08	2013-09	2013-10	2013-11	2013-12	2014-01	2014-02	2014-03	2014-04	2014-05
Lines Added	0	0	0	0	0	0	0	0	2,859	1,242	7,804	3
Lines Removed	0	0	0	0	0	0	0	0	40	271	6,420	2

Figure 8.7. Lines added and removed per month in project C3.



	2013-06	2013-07	2013-08	2013-09	2013-10	2013-11	2013-12	2014-01	2014-02	2014-03	2014-04	2014-05
Lines in Total	0	0	0	0	0	0	0	0	2819	3790	5174	5175

Figure 8.8. Number of lines per month in project C3.

Table 8.8. Contributions per Author in Project C3

Author	Commits (%)	Lines added	Lines removed
Punky Bruiser	28 (34.15%)	2739	978
Chaotic Ace	27 (32.93%)	6472	5485
Akhoch	10 (12.20%)	261	58
Ryan Kral	10 (12.20%)	3276	384
Aya	5 (6.10%)	651	481
Mentor	2 (2.44%)	7	1

Table 8.9. Effort Estimation for Project C3

Project	Staffing (person)	Schedule (month)	LOC	Ratio
C3	5	4	2,143	107.15

The system has a total of 2,143 LOC, where 1,257 were reused and 886 were newly added. Table 8.9 summarizes the comparison.

8.3.2. Software Metrics

Table 8.10 shows the software complexity metrics for the project. The metrics for the original Pac-Man video game, for project C3, i.e., the modified game, and the comparison are all presented together.

Table 8.11 shows the software structural metrics for the project. The rows show the metrics for the original Pac-Man video game, for project C3, i.e., the modified video game, and finally the difference between the two.

Table 8.10. Complexity Metrics for Original Pac-Man Video Game and C3

Package	**Files**	**LOC**	**DIT**	**NOC**	**Methods**	**CC**		
						Avg.	**Max.**	**Min.**
Initial system	11	1257	1	0	75	4.49	21	1
C3	14	2143	1	0	117	5.86	50	1
Difference	+	+	0	0	+	+	+	0

Table 8.11. Structural Metrics for Original Pac-Man Video Game and C3

Package	**FAT**	**H**	**Ca**	**Ce**	**I**	**A**	**|D|**
Initial system	22	2.09	0	0	1	0	0
C3	36	2.64	0	0	1	0	0
Difference	+	+	0	0	0	0	0

Finally, a graphical look at the interconnection of project components provides an overview of the extension of the systems. Figure 8.9 depicts the composition of the system.

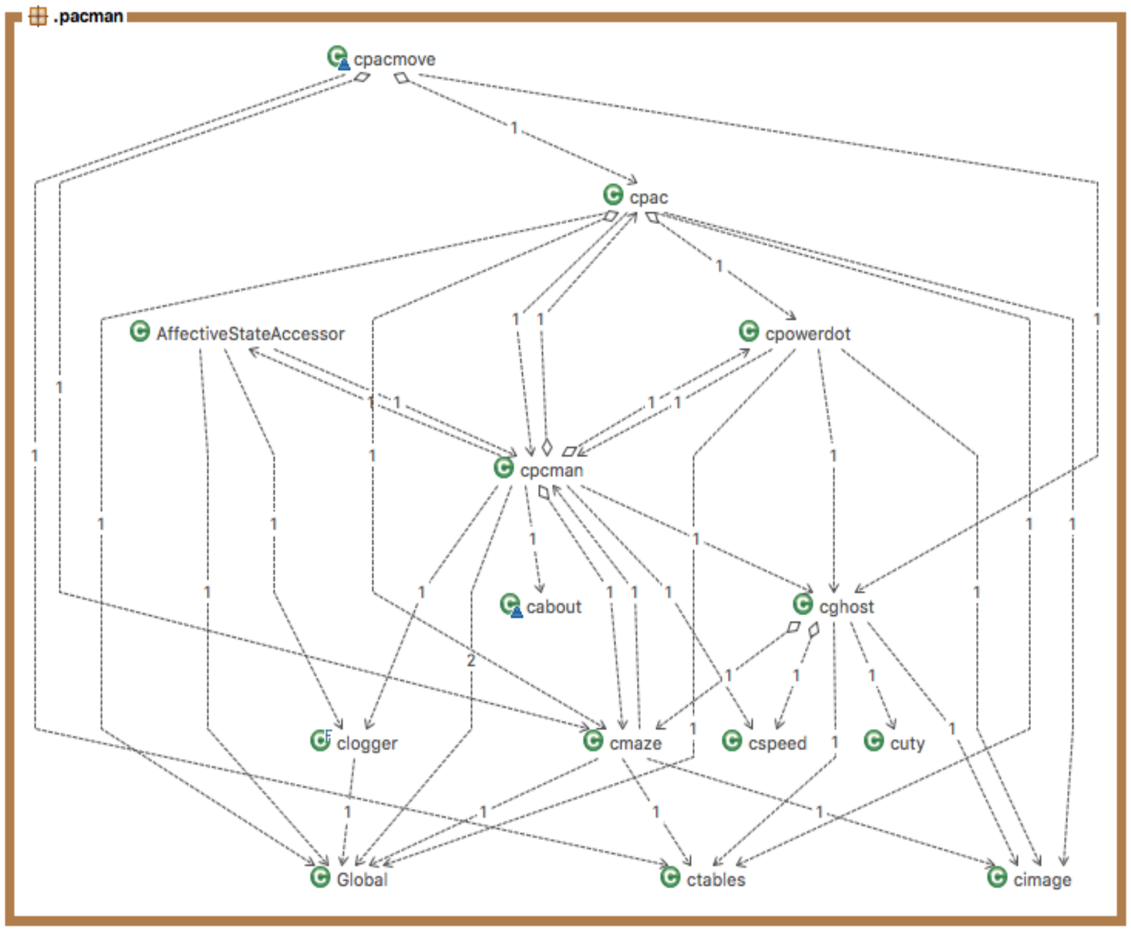


Figure 8.9. Dependency graph depicting the composition of project C3.

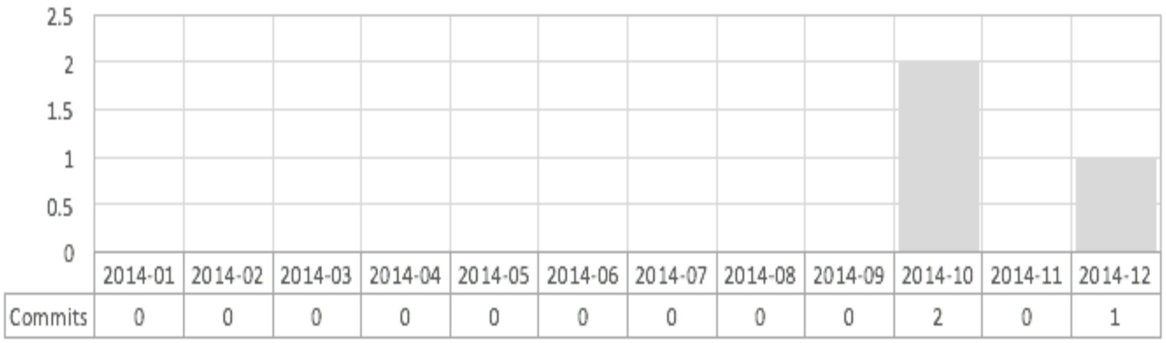
8.4. Analysis of Project C4

Project C4 built a system that corresponds to scenario B, as described in section 3.5.2, a multimodal affect recognition and adaptive system. Project C4 shows the feasibility of the scenario with a functional logic deployed separately from the adaptation logic. It built a functional logic in Unreal Engine 4, which is not considered for analysis, and an adaptation logic in Java, the analysis of which is described below. The code analyzed includes the final code for the adaptation logic and the testing tools created during the process to allow developers to become familiar with the process.

8.4.1. Process Measurements

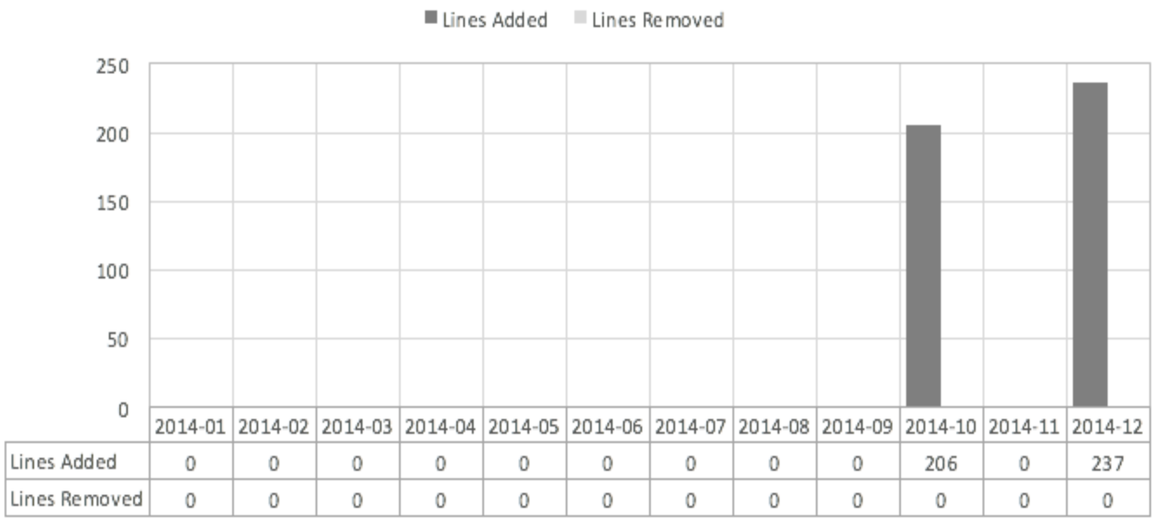
The C4 project was a two-semester project running in Spring and Fall 2014. The Git repository analysis could not be completed because the developers failed to make commits regularly. They only recorded 3 commits at the end of the Fall 2014 semester, as shown in Figure 8.10. Figure 8.11 shows the amount of lines added and removed per month. The repository had 443 lines. The historical growth of the number of lines stored in the repository is depicted in Figure 8.12.

The project was developed by a 5-member team. However, as shown in Table 8.12, only one author submitted contributions to the repository, i.e., participated in the Java interface. Therefore, all the code was attributed to that individual. Authors' names correspond to the usernames that students used to register in Git repository. 'Mentor' represents all the members of the mentoring team together. The students focused on the game's aesthetic, i.e., the Unreal Engine development, which justifies the small amount of Java code. Only the Java code is analyzed here – a module with 233 LOC having a simple adaptation logic and its interfaces. Table 8.13 shows estimated and real effort and schedule estimation.



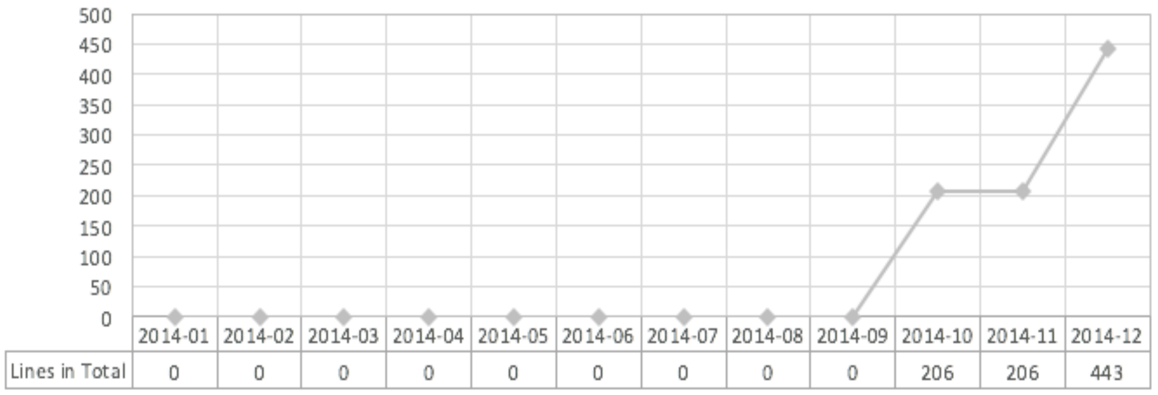
	2014-01	2014-02	2014-03	2014-04	2014-05	2014-06	2014-07	2014-08	2014-09	2014-10	2014-11	2014-12
Commits	0	0	0	0	0	0	0	0	0	2	0	1

Figure 8.10. Commits per month in project C4.



	2014-01	2014-02	2014-03	2014-04	2014-05	2014-06	2014-07	2014-08	2014-09	2014-10	2014-11	2014-12
Lines Added	0	0	0	0	0	0	0	0	0	206	0	237
Lines Removed	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8.11. Lines added and removed per month in project C4.



	2014-01	2014-02	2014-03	2014-04	2014-05	2014-06	2014-07	2014-08	2014-09	2014-10	2014-11	2014-12
Lines in Total	0	0	0	0	0	0	0	0	0	206	206	443

Figure 8.12. Number of lines per month in project C4.

Table 8.12. Contributions per Author in Project C4

Author	Commits (%)	Lines added	Lines removed
chornbeck	2 (66.67%)	439	0
Mentor	1 (33.33%)	4	0

Table 8.13. Effort Estimation for Project C4

Project	Staffing (person)	Schedule (month)	LOC	Ratio
C4	1	1.5	233	155.33

8.4.2. Software Metrics

Table 8.14 shows the complexity metrics for the project and Table 8.15 shows the software structural metrics for the project. A graphical representation of the interconnection of the project components provides an overview of the project extension and is presented in Figure 8.13.

Table 8.14. Complexity Metrics for Project C4

Package	**Files**	**LOC**	**DIT**	**NOC**	**Methods**	**CC**		
						Avg.	**Max.**	**Min.**
C4	4	233	1	0	23	2.43	8	1

Table 8.15. Structural Metrics for Project C4

Package	**FAT**	**H**	**Ca**	**Ce**	**I**	**A**	**|D|**
C4	1	0.4	0	0	**1**	**0**	**0**

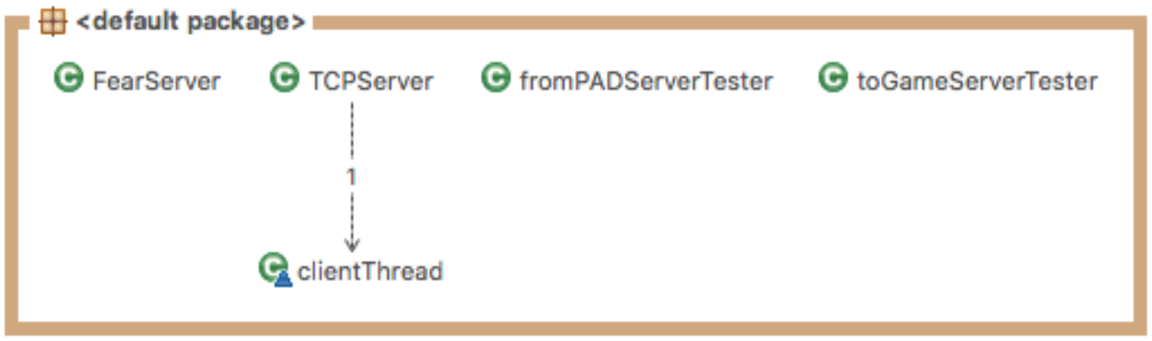


Figure 8.13. Dependency graph depicting the composition of project C4.

8.5. Analysis of Project R2

Project R2, AMT, built a system that corresponds to scenario C, as described in section 3.5.3, a multimodal affect recognition and adaptive system. It implements affect recognition and adaptation logics from scratch. The intention is to compare the results with another system using the ADAS affect recognition and adaptation infrastructure and processes. The composition of the Summer 2013 version of the AMT system is described in Gonzalez-Sanchez et al. (2014). The LOC reported here varies slightly from Gonzalez-Sanchez et al. (2014), as these are from the final version (Fall 2013). The Fall 2013 version has an increased number of functions and a decreased LOC due to refactoring. However, the structure remains the same.

8.5.1. Process Measurements

This was a 4-year project running from Spring 2009 to Fall 2013. A total of 15 developers and 4 managers were involved in different stages of the project. A team of at least 4 developers worked concurrently at every stage. Diverse versions of an intelligent tutoring system with affective companionship and meta-tutoring capabilities were released at intervals of 6 months.

The 4-year implementation process was managed using a SVN revision-control system, instead of Git; there was no particular reason except the preference for using SVN at that time for the team that began the development. The project comprises more than 1,600 revisions and 8 released versions. Differences between released versions include, among others, changes in requirements, enhancements in decision-making strategies, and bug fixing. The final release, with 22,979 LOC in 114 Java files, has an enhanced affective companion and refactoring improvements. Just for reference, the full project has more than 100,000 lines in more than 500 files, which include JPG, XML, TXT, CSV, PNG, HMM, BSK, form, GIF, JPG, properties, DLL, DRL, HTML, DAT and others formats. Figure 8.14 helps to contextualize each stage of production. In 2009 development was not the focus of the activities, in 2010 and 2011 development was related to the tutor module, in 2011 and 2012 to the meta-tutor module, and in 2013 (and late 2012) to the affective companion module (including affect recognition and the companion avatar). This final part of the project did not receive many development iterations.

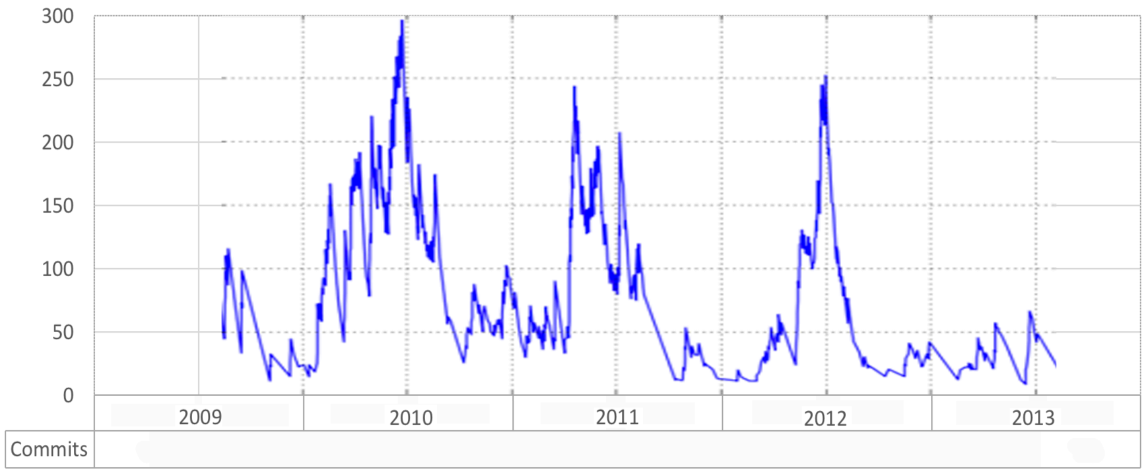


Figure 8.14. Commits per date for project R2.

The affective companion (AC) constitutes 4,838 LOC in 36 files. But, 574 lines are reused from legacy code that provides access to pressure, posture, and skin conductance devices. The effort calculated is shown in Table 8.16.

Table 8.16. Effort Estimation for Project R2

Project	Staffing (person)	Schedule (month)	LOC	Ratio

AC	4	12	4,838	100.79

8.5.2. Software Metrics

Table 8.17 shows the software complexity metrics for project R2. The system is composed of three main packages. The metrics are separated for each and a total presented in an additional row. Table 8.18 shows the software structural metrics for the project. The dependency graph provides an overview of the project extension and is presented in Figure 8.15. Only the AC component hierarchy is detailed inside the box; the other packages are shown as dots above the box. The structure of the legacy code appears at center of the box.

Table 8.17. Complexity Metrics for Project R2

Package	Files	LOC	DIT	NOC	Methods	CC		
						Avg.	Max.	Min.
tutor	54	15,418	1.02	0.03	920	3.34	78	1
MT	24	2,723	1.03	0.03	224	3.50	74	1
AC	36	4,838	1.10	0.10	360	2.83	54	1
Total	114	22,979	1.04	0.05	1,504	3.22	78	1

Table 8.18. Structural Metrics for Project R2

Package	FAT	H	Ca	Ce	I	A	|D|
tutor	288	5.35	23	13	0.36	0.03	0.60
MT	71	3.00	12	11	0.48	0.00	0.52
AC	75	2.11	11	12	0.52	0.06	0.42

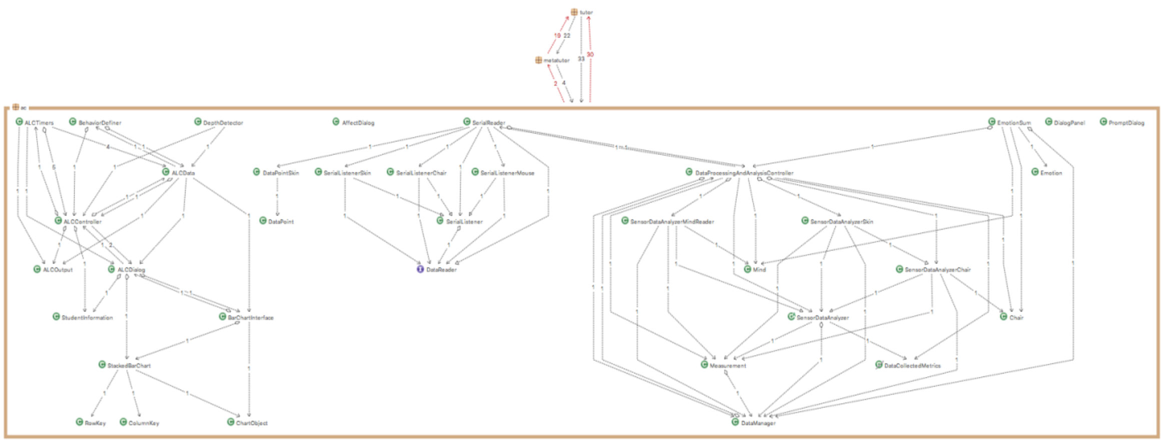


Figure 8.15. Dependency graph depicting the composition of project R2, detailing, inside the box, the affective companion package structure.

Since the AC was not built from scratch but based on legacy code from diverse sources, it is necessary to present the metrics of the legacy code. The values are presented below in Tables 8.19 and 8.20. Figure 8.16 shows a graphical representation of the interconnection of the components provided as legacy code. Notice the similarity with right part of the Figure 8.15.

Table 8.19. Complexity Metrics for Legacy Code Used to Create Package AC

Package	**Files**	**LOC**	**DIT**	**NOC**	**Methods**	**CC**		
						Avg.	**Max.**	**Min.**
Legacy AC	8	574	1.33	0.44	32	4.44	15	1

Table 8.20. Structural Metrics for Legacy Code Used to Create Package AC

Package	**FAT**	**H**	**Ca**	**Ce**	**I**	**A**	**|D|**
Legacy AC	14	1.87	0	0	1	0.12	0.12

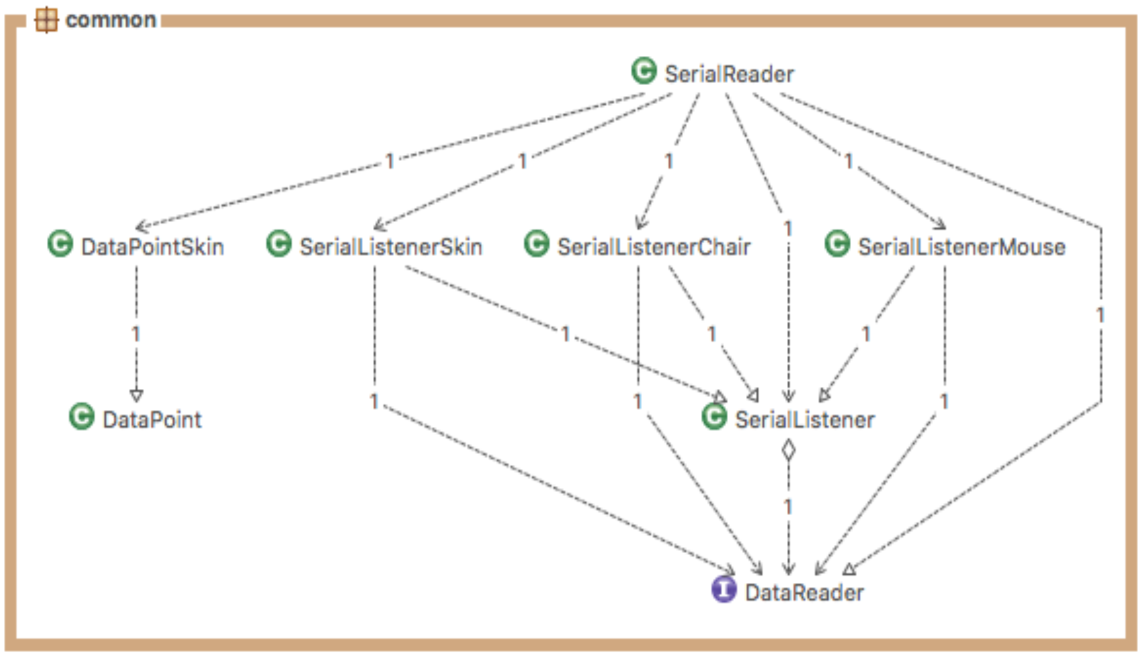


Figure 8.16. Dependency graph depicting the composition of the legacy code used to create the affective companion in project R2.

8.6. Analysis of Project R3

Project R3 built a system that corresponds to scenario C, as described in section 3.5.3, a multimodal affect recognition and adaptive system. It uses ADAS for the affect recognition and adaptation logics. It reuses a previous version of the AMT project, but:

a) removes the meta-tutor.

b) replaces the AC emotion recognition functionality with ADAS.

8.6.1. Process Measurements

This was a 2-year project, running from Spring 2015 to Fall 2016. It was developed by undergraduate students and led by a graduate student. It mainly focused on improving the tutoring and affective companionship capabilities. The goal was to improve and develop an intelligent tutoring system with affective companionship but omitting the meta-tutoring capabilities. One semester is dedicated to the reengineering of a tutor and one semester to the introduction of affect and adaptation. The final release has an enhanced affective companion and 1,101 LOC in a total of 4 Java files. The effort calculated is shown in Table 8.21.

Table 8.21. Effort Estimation for Project R3

Project	Staffing (person)	Schedule (month)	LOC	Ratio
R3	2	6	1,101	91.75

8.6.2. Software Metrics

Two comparisons can be made with the system produced for this project. The first relates to affect recognition, and compares the implementation of affect recognition in R2 and the implementation of affect recognition in the ADAS infrastructure. The second relates to the structure and complexity of the R3 itself and compares it with R2. Both Jess and Drools are external libraries, and the rules themselves are contained in separated files. The rule engine (Jess or Drools) does not affect the metrics for the tutor implementation significantly. Since Jess has been tested with ADAS, it was selected for use here. Table 8.22 shows the complexity metrics for the project. Table 8.23 shows the software structural metrics for the project.

8.7. Analysis of ADAS AR Infrastructure

As a reference for further discussion, the metrics for complexity and structure for each of the packages that form the ADAS infrastructure are presented. Table 8.24 shows the code complexity metrics for the ADAS AR infrastructure. In general, an average complexity project; even though maximum cyclomatic complexity in sensing and `toolkit.data` point out refactoring of a couple of methods is required in each package. The ADAS framework uses self-contained components that have high cohesion and low coupling. The cohesion is quantified by two metrics and the coupling by three metrics. Table 8.25 shows the structural metrics. Figure 8.17 shows a graphical representation of the interconnection of the components in each of the packages that form the ADAS infrastructure. The dependency graph facilitates an appreciation of its simplicity and its cohesion properties.

Table 8.22. Complexity Metrics for Project R3

Package	**Files**	**LOC**	**DIT**	**NOC**	**Methods**	**CC**		
						Avg.	**Max.**	**Min.**
tutor	46	8,363	1	0	562	2.81	66	1
ele	4	1,101	1	0	49	3.27	16	1
Total	50	9,464	1	0	611	3.04	66	1

Table 8.23. Structural Metrics for Project R3

Package	**FAT**	**H**	**Ca**	**Ce**	**I**	**A**	**|D|**
ele	3	1.00	15	27	0.64	0	0.36
tutor.audio	0	1.00	8	1	0.11	0	0.89
tutor.comm	2	1.00	12	2	0.14	0	0.86
tutor.data	8	1.12	16	6	0.27	0.25	0.48
tutor.dialogs	1	0.20	7	15	0.68	0	0.32
tutor.gui	7	0.88	5	12	0.71	0	0.29
tutor.log	0	1.00	14	6	0.30	0	0.70
tutor.model	25	3.25	19	25	0.57	0	0.43
tutor.parser	8	1.80	7	6	0.46	0	0.54
tutor.plot	0	1	1	4	0.80	0	0.20

Table 8.24. Complexity Metrics for ADAS AR Infrastructure

Package	**Files**	**LOC**	**DIT**	**NOC**	**Methods**	**CC**

						Avg.	Max.	Min.
sensing	12	461	2.08	0.92	44	2.05	**15**	1
perception	7	42	0.96	0	14	1.00	1	1
integration	3	94	1.67	0	16	1.19	2	1
introspection	2	33	1.00	0	10	1.00	1	1
behavior	3	44	0.67	0	4	1.50	2	1
engine	4	51	1.00	0	13	1.08	2	1
toolkit.comm	2	143	1.00	0	12	2.58	8	1
toolkit.net	3	204	1.33	0	18	2.17	8	1
toolkit.data	13	608	1.08	0.08	91	1.99	**46**	1

Table 8.25. Structural Metrics for ADAS AR Infrastructure

Package	FAT	H	Ca	Ce	I	A	|D|
sensing	15	1.33	15	14	0.48	0.33	0.18
perception	4	**0.71**	3	1	0.25	0.43	0.32
integration	1	**0.66**	1	3	0.75	0	0.25
introspection	1	**1.00**	3	0	0	0	**1.00**
behavior	2	**1.00**	1	5	0.83	0.33	0.17
engine	4	**1.25**	3	3	0.5	0.25	0.25
toolkit.comm	1	**1.00**	1	5	0.83	0	0.17
toolkit.net	1	**0.66**	14	2	0.12	0	**0.88**
toolkit.data	19	1.53	24	0	0	0.08	**0.92**

8.8. Summary

Process measures and software metrics for the quantification of the complexity and structure of the six selected projects were presented in this chapter. In addition, the metrics for the ADAS affect recognition logic are included for the purpose of comparisons with projects C1 and R3. Comparisons of these findings and a discussion of their relevance to implementing affective adaptive systems using the approach proposed in this dissertation are presented in the next chapter, supported by the data presented here.

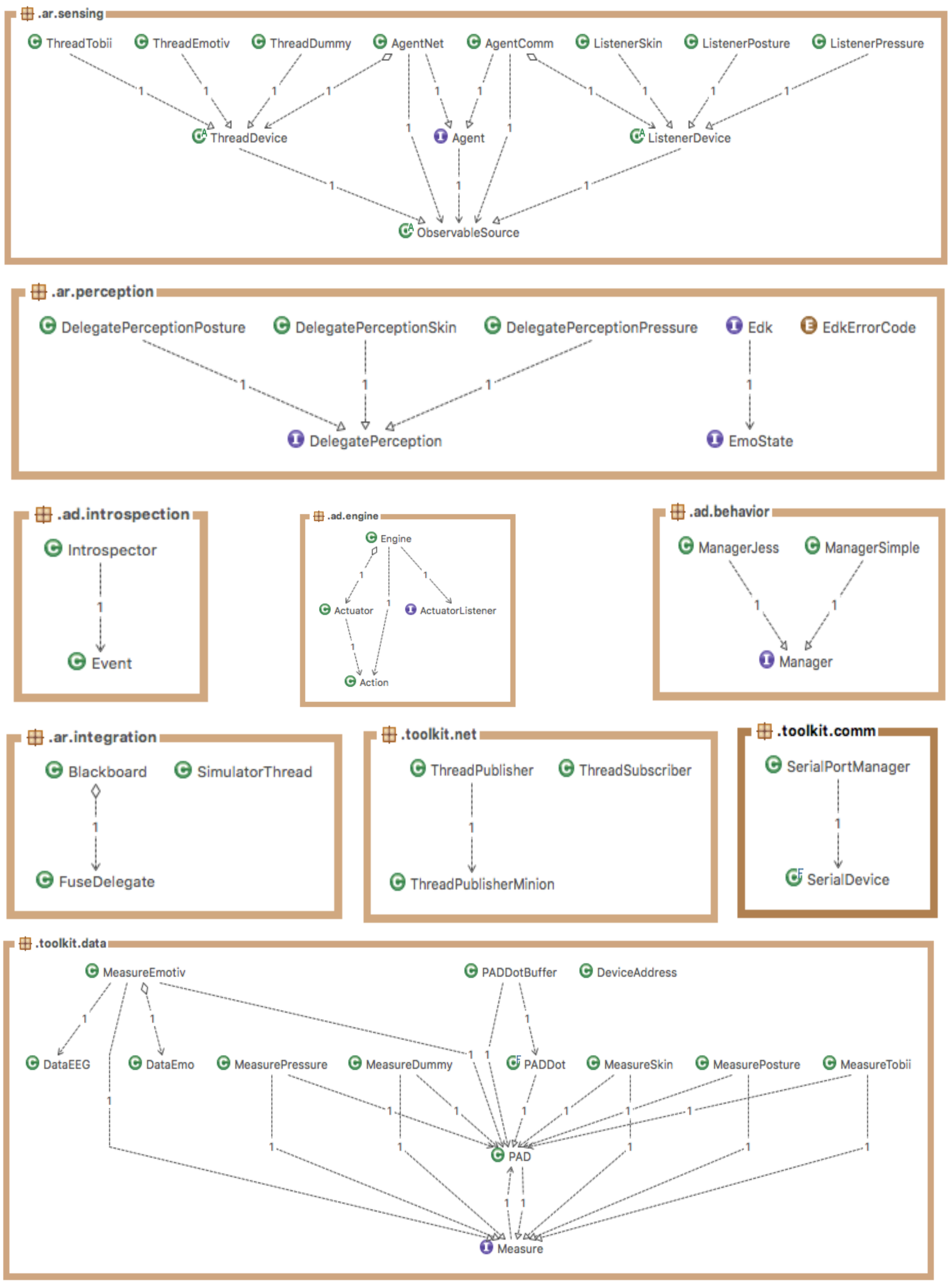


Figure 8.17. Dependency graph depicting the composition of ADAS packages.

CHAPTER 9
EVALUATION AND DISCUSSION OF RESULTS

The empirical evaluation of ADASPL entails an in-depth demonstration of the SPL approach by applying the common and reusable architecture, infrastructure, and processes, while customizing pieces for representative case studies. The empirical evaluation of the SPL approach is both qualitative and quantitative as it involves:

a) the validation of achieved results and the products created, under diverse conditions.
b) the validation of the cost-effectiveness of the engineering effort required to achieve the results.
c) the validation of the potential capabilities of the SPL assets (architecture and infrastructure) and understandability of the process.

For each of these points, a claim was made and the case studies allow the claim to be evaluated, as discussed in the following sub-sections of this chapter. The analysis of the achieved results in terms of product usability is also commented upon.

9.1. Claim of Generality for Broad Deployment

In Chapter 1, the relationship between generality and the use of a model and an infrastructure is described. Both the model and the infrastructure impact on improving generality. They support the assembly of a broad range of products with diverse functional dimensions. Products are created and then assessed in terms of algorithmic and structural complexity. The software quality attributes of reusability, flexibility, extendibility, and maintainability are evaluated for each product using software metrics.

The purpose of this evaluation is to assess the success of the assembly process for quality products, more specifically whether the infrastructure provides the desired functionalities in a relatively convenient way, while allowing flexibility in the selection of sensing devices, fusion methods, and adaptation algorithms and maintaining system quality attributes.

The evaluation of the developed products has two aspects:

a) demonstrating the quality and functionality achieved through the application of the architecture and infrastructure to create sample applications that show how to use and customize the infrastructure. For practical reasons, this evaluation targets a representative subset of systems and their functional dimensions. The evaluation was performed with representative systems (C3, C4, C5, C6, and R1), and their quality was measured using complexity and structural metrics; these measures enable a discussion of the necessary trade-offs.

b) providing an in-depth demonstration in a complex real-life system; I use an intelligent tutoring system, operating in the science-learning field. A comparison is made between the systems developed with (R3) and without (R2) the use of the ADASPL approach.

9.1.1. A Set of Sample Applications

A key aspect of the demonstration of generality is the application of various parts of the ADASPL approach under diverse conditions; in this chapter, generality is demonstrated through the implementation of 9 systems. The systems were developed under the following diverse conditions:

a) Developers: teams of 3 to 6 CS, CSE and informatics BS students.

b) Goals: virtual world avatars, intelligent companions, games, and tutoring systems.

c) Languages to implement functionality: Java, but also C++, C#, and interfaces with Unreal Engine 4 and Unity.

d) 3 scenarios with the architecture covering affect awareness and affect adaptation, as well as monomodality and multimodality.

Of these 9 systems: 2 failed to be completed; 3 were submitted as demos or posters to conferences on gaming and user interfaces, of which 2 were accepted for presentation and publication; 1 was a 4-year NSF-funded project used to run several studies and generated diverse publications; and 1 is to be used as a tool in a PhD dissertation.

The 9 systems represent 3 groups of closely related projects used for analysis and to assess the quality of the results achieved: the first is represented by C4 and includes C6 and C5; the second is represented by R1 and includes C7 and C2; and the final group is represented by C3 and also includes C8. From the metrics in Tables 8.7, 8.8, 8.11, 8.12, 8.15, and 8.16 in Chapter 8, the following observations can be made for each of these groups:

a) R1: simple affect awareness in one file with 181 LOC. All metrics sustained a low complexity level. The average cyclomatic complexity of 8.80 is due to the functionality implementation (if-else conditions triggering events). Simplicity is also supported by perfect structural metrics.

b) C3: extension of an affect aware video game based on legacy code. There was an obvious increment in files and methods, but this did not correspond to a considerable increment in LOC. It seems that a decision logic to trigger events in the game logic was implemented and nested in a small number of methods, which would explain the maximum cyclomatic complexity of 50, and therefore the increment in the cyclomatic complexity average. Even though a cyclomatic complexity of 50 is a red flag (it exceeds the recommended limit of 10 and even the warning limit of 20), it is not due to the inclusion of the ADAS infrastructure, but rather due to the developer's ability in implementing the functional logic. The game uses ADAS for affect recognition but not for the adaptation logic. Regarding structural complexity, FAT and H increase but remain within the acceptable range, below 60 and in the range of 1.5 to 4.0, respectively. The slight increase from the original is justified by the added functionality. As expected, distance, stability, and abstractness do not change.

c) C4: a game built from scratch using a 3D engine. A small amount of files and a low LOC were required. to interface ADAS with the game engine. Game-engine coding effort is not considered here. The small system justified the low DIT and NOC: a figure of 23 methods with an average cyclomatic complexity of 2.43 is considered acceptable. The refactoring of one method that had a cyclomatic complexity of 8 would be desirable. Structural metrics indicate that the package has low cohesion, probably due to its simplicity, and is therefore fully unstable (easy to change) and concrete.

In general, starting from scratch, modifying legacy code, using one common programming language, or even linking with external engines did not create a complex system. In all circumstances system metrics were acceptable and the complexity related to affect recognition is well isolated from the system functionality.

<u>9.1.2. Comparison of Real-Life Systems with and without ADAS Infrastructure</u>

The comparison of R2 and R3, real-life systems with years of development and a diverse and extended team, based on the metrics shown in Tables 18, 19, 20, 21, 22 and 23 in Chapter 8, suggests the following:

a) The original system complexity reflects years of work; the complexity of the methods is manageable with an average cyclomatic complexity of 3.5. But the system has cases that reflect a poor partition of responsibilities in general. However, the AC package deserves attention here as having the lowest complexity of the three. Problems with this system in fact derive from the structure more than the complexity of the programming. The structural complexity metrics for the AC have a FAT value of 76 (above the limit), and even though H is within the accepted limits, the structure in Figure 14 reflects a lack of architectural design. The high distance (0.42) can be attributed to the lack of abstractness, even though instability is balanced.

b) To create the AC in R2, 32 methods in 8 classes were provided as legacy code for reading sensing devices. The legacy code's cyclomatic complexity is acceptable (4.44). The legacy code's structure is well-established, and the encapsulating package is non-FAT with H of 1.87. The 0 values in metrics Ca and Ce for the legacy code imply that it is not being used (i.e., is not connected with other packages, for instance, a functional logic). Thus, instability and distance are not really significant when measured for the package alone. Focusing on complexity, note the increment from a maximum cyclomatic complexity of 15 in the legacy code to a maximum cyclomatic complexity of 54 in the AMT AC. Even though the average is lower for the AC (2.83), this can be attributed to an increase in the number of methods from 32 to 360.

The legacy code supported affect recognition, which is a significant part of R2, AC, but not its full functionality. The companion, its intelligence, and its integration with the tutor and meta-tutor form a complex system. Numbers show that the difficulties in the AC are not inherited from the code used to read the sensing devices.

c) R3 uses the tutor module of R2, removes the MT modules of R2, and reengineers the AC using ADAS. The numbers for the AC engineered for R3 show a slight improvement in code complexity and a noticeable improvement in structural complexity. So, the new approach does not affect the functionality, nor does it increase the size or code complexity, but it does improve the structural quality. The new structure has low coupling and high cohesion – a key strength for understandability, extensibility, and maintainability.

9.2. Claim of Cost-effectiveness of Engineering Effort

The cost-effectiveness of the engineering effort required to use ADASPL is an important validation, as it establishes that the traditional approaches to developing new systems requires significant duplication of effort and, consequently, higher costs compared with the use of ADASPL. Chapter 1 described the relationship between cost-effectiveness and having an infrastructure and process guidelines. Both the infrastructure and the process guidelines impact on improve cost effectiveness.

Measuring the engineering effort to develop a software system, on an absolute scale and independently of human factors, is a challenging task. It is dependent on the ease-of-use of the approach, its understandability, and the potential reusability of the approach's technical solution.

To quantitatively and qualitatively assess cost-effectiveness requires the following tasks:

a) demonstrate that the approach saves time and development effort for engineers by characterizing the self-adaptation tasks and providing a coarse-grained, task-based estimation of effort.

b) assess the effort savings with the proposed approach relative to current practices.

To show effort savings,

a) I characterize the development tasks and provide a coarse-grained, task-based estimation of effort (for C3, C4, and R1). I then compare these with the estimations calculated by an algorithmic software estimation model based on current practices.

b) I qualitatively assess and evaluate the effort savings for self-adaptation by comparing the implementation of a complex real-life system with and without the proposed approach.

Each of these is discussed in detail below.

9.2.1. Capstone Team Effort versus Estimation Effort

An immediate success can be claimed in that of the seven capstone teams involved in applying the approach, all successfully achieved partial or full completion of their projects with the desired capabilities, and the two that did not succeed in doing so appears to have experienced team management problems rather than issues with the project itself. Now, let us consider the details regarding time savings and ease-of-use as relevant to the understandability of the approach.

Chapter 8 provides information regarding the time frames and deployments of the projects used to test this approach. From the information in Tables 8.8, 8.12, and 8.16 it is possible to identify the following:

a) C3 is estimated to take 5 months, with an effort of 2.6 person-months, to produce a total of 2,143 LOC. This capstone project was scheduled for an entire year, but the development logs show that it was realized over 4 months by 5 developers. However, it was observed that 2 developers made 67% of the commits. Therefore, arguably, C3 could be a 4-month project completed with an effort of 3 person-months.
b) C4 is estimated to take 2.8 months with an effort of 0.4 person-months to produce 233 LOC. This capstone project was scheduled for 1 year, but the development logs show that it was realized over 2 months by 1 developer (omitting the game environment, and just considering the game logic and its interface with ADAS). In this case, reality closely resembles the estimation.
c) R1 is estimated to take 2.5 months with an effort of 0.3 person-months to produce 181 LOC. This project was realized during 1 semester; it was not formally measured but reported to have been completed in 1 month by 1 developer. The reality seems close to the estimation.

In these three cases, it can be said that the time taken to develop this kind of project using the ADASPL approach does not differ from the estimation of similar projects in other contexts and domains. The application of the ADASPL approach does not add complexity to the implementation of functionality, although the estimated functionality includes interfaces and code related to connection with the ADAS infrastructure. Moreover, it removes the effort related to affect recognition and its inclusion in the project.

9.2.2. Research Staff Effort with and without ADASPL

A comparison of the effort needed to implement affect-driven adaptation capabilities using ADAS with those needed for the same task using a different approach is measured in real-life research projects R2 and R3. The effort required for R2 and R3 is compared below.

Project R2

It is important to remember that R2 implementation has fixed sensor devices, and a synchronization approach and emotional state model are implemented. Metrics indicate a reduced development time in R2 compared with an estimated for a project with similar characteristics. Arguably, the structural metrics could suggest that the quality of the software design was sacrificed to deliver the project on time.

The AC functionality (to improve learning) was not fully achieved (VanLehn et al. 2014), which may have been due to limitations in the affect recognition performance. Related research has produced positive results with similar approaches (devices and models) (Arroyo et al. 2009).

Notice that in Figure 8.14 the number of commits (i.e., work activity) related to the AC (June 2012 to July 2013) is ineffectively low compared to the commits related to the creation of the tutor and meta-tutor. As a whole, a low level of commits does not necessarily indicate fewer LOC, but it relates to the collaboration process behind the composition of the code, especially considering the fact that different developers were involved.

Project R3

The effort for R3 was not measured for all steps. However, it begins with an earlier version of R2 and makes the following changes:

a) it removes the meta-tutor;

b) it removes the AC;

c) it fixes memory leaks in the tutor;

d) it replaces the AC with ADAS with the intention of including new sensor devices (Emotiv headset and eye-tracking); and

e) Drools is replaced by Jess as a result of the introduction of ADAS.

The data in Table 8.20 corresponds with our expectations based on these changes. A reduction in LOC is expected, and at least 574 lines from the legacy AC are removed, as well as the integration and emotional state modeling code, but, the size of the AC stays large. This is perhaps due to its GUI. However, the metrics in Table 8.21 show substantial improvement in structural complexity. In summary,

a) there is no substantial difference in terms of code size, and.

b) there is a significant difference in terms of potential extensibility and modifiability.

Therefore, although the data did not indicate a substantial contrast, the real impact of these projects relates to the cost of future changes. For instance, changing the emotional state model, using new sensing devices, or changing inference algorithms can be expected to have a lower cost in R3 than in R2.

Adaptations run with the Drools rule engine achieve similar results as those run with Jess. The separation of the adaptation rules from the system is a positive feature of both R2 and R3.

Finally, regarding understandability, both the team developing R2 and the team developing R3, were knowledgeable of affect recognition, adaptation, and rule engines topics. Thus, models and patterns to support understanding of the field cannot be evaluated for these projects.

9.3. Claim of Feasibility

In relation to feasibility, I focus on the operational factors showing ADASPL as a dependable, low-risk, high-payoff practice, particularly intended for use by engineers with limited experience. Chapter 1 described the relationship between feasibility and having a model and an infrastructure. Both the model and the infrastructure impact on improve feasibility. Operational feasibility is inherently related to:

a) understandability, i.e., enabling engineers to use the architecture, customize the framework, and follow the guidelines. It abstracts away fine-grained details, and in the process allows domain experts to define adaptation choices and compose adaptation strategies.
b) the use of high quality tools, i.e., an architecture that guides software engineers in the process targeted to achieve composability, and an infrastructure targeted at manufacturing diverse kinds of systems provides reusability.

On the one hand, regarding understandability, a qualitative evaluation shows whether the SPL is understandable. I assessed the expressiveness of the approach and the way it facilitates an understanding of how to engineer affect recognition, affect awareness, and affect-driven self-adaptation capabilities. The assessment was accomplished by surveying developers and qualitatively analyzing their achievements. Particularly, analyzing whether the architecture and patterns provide constructs to specify strategies for different requirements, support to combine those constructs meaningfully in diverse dimensions, and mechanisms to automatically select and carry out adaptations that integrate strategies for achieving multiple objectives. This evaluation collected data from the developers' experiences to inspect the balance between the simplicity and the power of the framework, i.e., validating whether the practice is low risk and high payoff with the provided resources and guidance.

On the other hand, regarding assets, an in-depth formal analysis of the architecture is not performed due to time constraints and the fact that the architectural model is an adaptation of related models applied in the self-adaptation field, but an empirical analysis is run. This analysis of the infrastructure evaluates its functionalities and qualitatively demonstrates its capabilities. The infrastructure validation considers factors inspired by Cheng, Garlan, & Schmerl (2005), who evaluate assets for self-adaptation driven by resource-related issues. The evaluation of the infrastructure measures its complexity and structure, as well as comparing it with an attempt at implementation from scratch.

9.3.1. Understandability of Process as a Result of Architecture and Patterns

The developers' understanding of the architecture, framework, and guidelines was surveyed and found to be fair, and its integration with libraries, engines, and other frameworks found to be feasible.

To evaluate whether the process was understandable, developers were asked to explain it. Each team was asked for a group presentation and a video. They were given the following specific instructions:

a) Make a final 15-minute presentation on the project to entire whole class. One constant in every team's presentation was the use of a slide to present the stages of the affect recognition and adaptation logics as separate from the logic of their project. Another constant was the use of one slide to explain their understanding of the PAD emotional state model as a cornerstone of how emotions are combined.

b) Make a 1- to 3-minute video explaining their project. In this video they black-boxed the complexity of the process, mostly showing the hardware they were using and presenting ADAS as a platform that provided them with what they needed to deal with affect and emotion representation.

Figure 9.1 shows examples of how students modeled the process. They understand the process and explain it in a public presentation. Sensing, adaptation, and system functionality are clearly separated and intended to be explained as communicating entities.

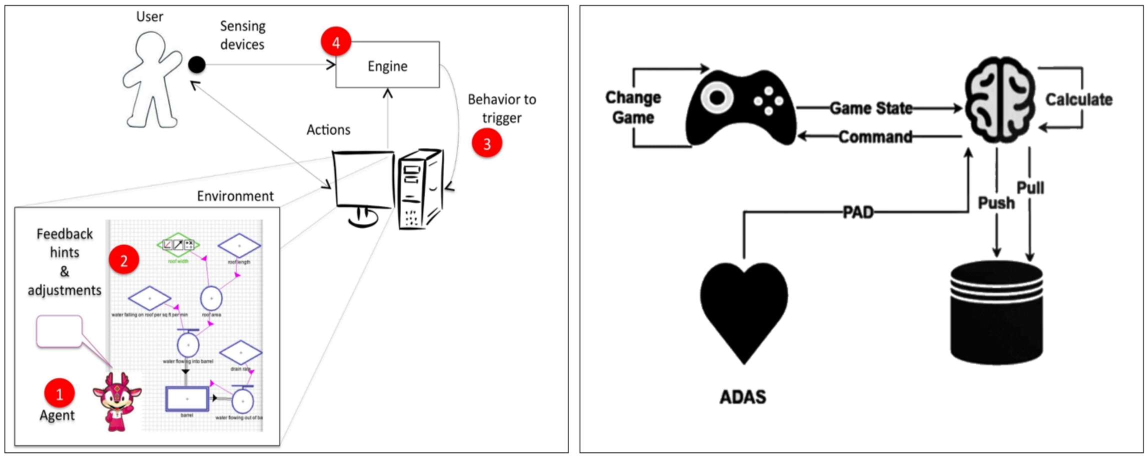


Figure 9.1. Student team slides: (a) an affective companion for a tutor; (b) a video game environment.

9.3.2. Quality of Infrastructure

Complexity. The ADAS infrastructure provides functionalities manufactured through several interactions. The complexity metrics from Table 8.24 in Chapter 8 show:

a) Packages are below the suggested threshold of 1,000 LOC per class. Packages with a relatively high LOC are: (1) the package that provides encapsulation for data handling (608 LOC), where several LOC are from getter and setter methods, constructor methods, and auxiliary methods, and (2) the sensing package (461 LOC), which implements the hardware devices in use. These two packages represent a substantial portion of the work that the framework alleviates.

b) DIT and NOC are below the thresholds (2-5 and 0-10, respectively), which for a framework seems acceptable. It should be taken into account that the extensibility goal of the framework keeps it in low values for these metrics, so that when used (extended) the target system's metrics are not jeopardized.

c) Cyclomatic complexity mostly stays below the suggested standard threshold of 10. It is relevant to notice that the complex packages are those providing key functionalities: reading physical ports for data collecting (2.58), networking (2.17), and data handling (1.99) and the sensing package, which handles sensor devices (2.05). The complexity of these key task implementations is known and has been described in previous chapters. The maximum complexity for the sensing and data packages (15 and 46, respectively) suggests that methods in each package might require refactoring.

Having a low complexity and good understandability proves the framework's potential to be reused, extended and modified.

Complexity in the infrastructure is compared with complexity of C1 as follows:

a) C1 and ADAS AR have the same amount of files even though C1 implements less functionality than ADAS and ADAS has a higher number of LOC.

b) DIT and NOC are higher in ADAS but not significantly. They are acceptable for both systems.

c) The number of methods in ADAS is higher in accordance with the increase in LOC and increased functionality implementations.

d) Average cyclomatic complexity is slightly higher in ADAS, clearly raised by a particular method that could be refactored (represented by the maximum value 46).

Complexity in general is closely similar in both systems; however, C1 implements less functionalities than the ADAS AR.

Structure. The structural analysis of the infrastructure shown in Chapter 8, Table 8.25, shows:

a) The FAT levels of the infrastructure packages are far below the limit (60). The FAT level is so low that the dependency graph for each package is simple and readable, as shown in Figure 8.17 in Chapter 8. The packages are slim point in favor of low complexity, maintainability and understandability.

b) The average H is below the suggested lower threshold (1.5 - 4.0). However, for the packages `sensing` and `toolkit.data`, H is near the lower threshold of 1.5. The other packages are mostly interfaces or abstract classes, and so the metric does not apply to them. It can be said that cohesion is satisfactory in general.

c) Ce and Ca highlight the large amount of classes that the sensing package depends on and the large number of classes using `sensing`, `toolkt.net`, and `toolkit.data`.

This seems normal given that `sensing` is the starting point for the process, `toolkt.net` is the foundation for communication in the agent-based architecture, and `toolkit.data` supports every step in the process.

d) Even though the recommended threshold for D, 0.1, is surpassed by all packages, they remain close to the ideal main sequence, with the exceptions of `toolkit.data` (0.92), `toolkit.net` (0.88), and `introspection` (1.00). Looking at I and A for these three cases shows them to be almost 0. This puts those packages in the pain zone. It suggests the need for refactoring to increment A, since the high Ca value of these two cannot be avoided.

Overall, the infrastructure has acceptable cohesion, is well suited to internal coupling, and has potential for good external coupling. The structural complexity is well-suited for a framework, even though refactoring could arguably slightly improve the metrics to put them closer to the ideal thresholds.

Moreover, comparing the ADAS affect recognition metrics with the metrics of the attempt at implementing affect recognition from scratch (project C1), it can be seen that:

a) FAT problems with C1 are evident in Figure 8.4 of Chapter 8, particularly the `myStuff` package, which seems to have the full implementation of C1 functionality. The increase in FAT levels for both projects clearly indicates a warning regarding C1's structure. However, H stays within the recommended limits, suggesting an over-populated package (several relevant classes in one container) instead of a dependency problem.

b) Notice that most of C1 is stable while most of the ADAS affect recognition is unstable. Thus, ADAS affect recognition is less resilient to change. Note that the pain zone (A=0, I=0) for ADAS is strongly enclosed in the packages related to networking and data handling, while C1 fails to show a defined structure and encapsulates almost everything in a single package. C1 extensibility and maintainability is complex.

The structural complexity of C1 shows resilience to change, limited modifiability and therefore reduced reuse possibilities. This demonstrates the importance of the ADAS affect recognition implementation and that its use represents a good choice for developers, even though the ADAS framework itself could benefit from the refactoring of localized classes and methods.

9.4. Product Quality and Usability

Since there is a considerable amount of work showing that affective support is good, I only show that SPL products correspond to those that have been shown to be beneficial. An evaluation of the users' experiences need to be analyzed carefully since several factors are involved: the users' experiences are in part due to the functionality offered by the product, i.e., software components external to the SPL, and in part due to the qualities and operation of the assets provided by the SPL and the accurate application of the processes and guidelines by the developers involved.

I argue that the manufactured products are along the lines of what has been shown to be beneficial.

Evaluating the user experience entails measuring the following:

a) Product performance. This related to whether the timing of both the adaptation of the target system and the monitoring of affect is able to show results that make the user "feel" the computer understands his/her feelings in real-time. It is important to consider that, if the adaptation or the affect measurements undergo a delay, the achieved adaptation will not be relevant anymore. This evaluation is informal and obtained by surveying a set of users.

b) Overheads caused by adding ADAS implementation to the execution of the target systems. The behavior of a systems in an affect-driven self-adaptive version and a traditional non-adaptive and non-affective version were compared.

User surveys suggest that for the specific population used in the experiment, the user experience caused by self-adaptation and affect measurements achieved in the products created with the proposed approach positively impact human-computer interaction, i.e., the use of affect to drive adaptation within the framework components is shown to be viable in diverse contexts with the integration of diverse sensing devices. This is specifically evidenced by,

a) Virtual world avatar demos at ACII.
b) Modified Pac-Man video game demos at UIST.
c) Three gaming systems tested and demonstrated as capstone projects between Spring 2013 and Fall 2015 and two more at the end of Spring 2016.

Finally, the reviewer's comments on the papers submitted by students to conference demos included the following:

- *"Having such a full multimodal framework is clearly the main strength of the proposed system and as such it deserves to be presented as a demonstration."*
- *"An interesting demonstration of a multimodal framework; I suggest authors to dedicate some time at the demo to present the framework and its potential."*

9.5. Summary

A suite of 9 systems were developed, each applying ADASPL. Only 2 failed to achieve its objectives; 3 were submitted as demos or posters to conferences on gaming and user interfaces; and 2 were accepted for presentation and publication. One was a 4-year NSF-funded project used to run several studies and generate diverse publications. An immediate success can be claimed in that of the 7 capstone teams involved in applying the approach, all successfully achieved partial or full completion of their projects with the desired capabilities, and the group that did not succeed in doing so appears to have experienced team management problems rather than issues with the project itself.

Evaluations discussed here included: (1) applying the approach to manufacture several systems; (2) quantifying the quality of these systems; (3) reviewing the time frames and deployments for the projects; (4) comparing the effort needed to implement affect-driven adaptation capabilities using ADASPL with that needed for the same task using a different approach; (5) an informal evaluation surveying developers' understanding of the process; and (6) an informal evaluation surveying the users' experiences that suggests that the achieved results positively impact on the human-computer interaction.

CHAPTER 10
CONCLUSION AND FUTURE WORK

In conclusion, the data shows that the ADASPL approach has promise. The research hypothesis stated that the proposed approach makes it feasible to include affect-driven adaptive capabilities while helping to reduce engineering effort, with a reduction in the perceived complexity, for assembling generalized components. And, results confirm this feasibility and suggest a decrease in development effort compared to previous experiences; developers who applied the ADASPL approach completed a broad set of systems in the affect-driven self-adaptation domain on time in almost all cases, and they were able to focus on the system functionality rather than the complexity of implementing an infrastructure for affect recognition or self-adaptation. Furthermore, applying ADAS reduced testing time. Broadly, it can be stated that the use of the ADAS framework through ADASPL favors the quality attributes of understandability, reusability, flexibility, extendibility, and maintainability. However, limitations also surfaced and are important to mention, they present opportunities for improvement and can be used to prioritize future work.

10.1. Promising Results

Our evaluation suggests that it is feasible to achieve cost-effective, affect-driven self-adaptation using generalized assets; although, it is worth noting that ADASPL rests on some important assumptions: (1) it is assumed that sensing devices and their software drivers were available and configured in the object machine, or the developers received support for their installation in their own computers; and (2) it is expected that the developers' expertise included threading and networking communication flow (TCP/IP sockets), software design patterns, and that they were knowledgeable in object-oriented programming concepts of interfaces, abstract classes, and overriding.

Generality. The results from case studies run with ADASPL seem favorable to affirming that it is possible to generalize the assets and processes dealing with the inclusion of adaptation driven by human factors, such as affect. Products assembled with the ADAS infrastructure following the ADAS architecture show exceptional results when measuring structural qualities and do not cause an increase in code complexity when added to existing systems nor in new systems compared with implementations from scratch following other approaches. For researchers in the area, ADASPL supported the development of a research application that compared with a similar one developed without the ADASPL approach showed improved structural characteristics and therefore flexibility, extensibility, and maintainability.

Feasibility. The results of having developers using ADASPL indicate the feasibility of the ADASPL process, i.e., that it is easily accessible for even inexperienced developers to use it in the long-term. ADASPL provides foundational common tasks and structural composition allowing developers to better take advantage of new technologies and focus on exploring their creativity. For novices in affect and adaptation (undergraduate senior students), ADASPL provides the opportunity to understand and practically apply affect recognition and affect-driven adaptation. Further, ADASPL helped developers to perceive HCI features (such as affect recognition capabilities) as not being overly complex.

Cost effectiveness. Measuring the development time and the size of the project developed shows that effort is comparatively lower using this approach than developing from scratch. This represents an advancement in the field since building affect-driven self-adaptive system such as the ones outlined in Chapter 7 are a costly proposition if important components such as the affect monitoring, adaptation, and translation mechanisms have to be built from scratch, particularly when there is no guideline indicating where to start or how to proceed.

User experience. Moreover, after testing their products with users, developers informally surveyed the users and received positive responses stating that adaptation and affect recognition represent improvements in the user-experience and the system's potential to achieve its goals. Empirical results suggest that ADASPL helped developers to manufacture enhanced products, especially designs that seek to improve user experience. However, it is important to mention that a possible source of bias in the developer experience could relate to the excitement of senior students learning about HCI concepts and using state-of-the-art devices. Even though it does not change the feasibility results, it creates a bias in the engineering process and perhaps also in the user experience.

<u>**10.2. Future Work**</u>

Future work to be considered in the short term includes the following:

a) Framework improvements will need to be implemented. As in any framework or library, iterative improvements are part of the software development life cycle. The most important quality to be addressed is performance.

b) A memory-leak was reported, which caused problems after the prolonged operation of the affect recognition threads (after approximately one hour).

c) Moreover, a refactoring could improve the ADAS quality attributes.

d) The complete developer's documentation is to be published online.

In the medium term, future work consists of addressing the following in the affect recognition logic:

a) *Sensing*. There are opportunities for future work on the sensor-computer integration. So far, the channel to integrate them uses a USB or serial port, or realizes a network connection. All sensors in the suite included in ADAS framework satisfied this requirement and extensions to include new sensors are possible as long as they use these connection channels.

b) *Perception*. There are opportunities for future work on the incorporation of additional inference algorithms in the ADAS framework. Inference algorithms included in ADAS framework encompassed these for the in-house sensors, described in Chapter 2, and interfaces to the native algorithms provided with commercial off-the-shelf devices.

c) *Integration*. There are opportunities for future work on the incorporation of additional synchronization and fusion algorithms in the ADAS framework. The synchronization method included in ADAS framework is a state-machine method which, instead of considering an affective state to exist only in the period of time equivalent to the sample, assumes that the state persists until a new state is obtained. The fusion method included in the ADAS framework maps each affective state measurement to a vector in the 3D emotional state model and adds up all of the vectors.

d) *Synapse*. There are opportunities for future work on the incorporation of additional data streaming to the ADAS framework. Synapse functionality in ADAS is provide with TCP connections among agents and the central control unit and among the central control unit and the adaptation logic. TCP provides a point-to-point channel for applications that require reliable communications. However, the reliability of TCP causes performance degradation and may hinder the usefulness of the service. Other approaches, such as UDP, can be explored.

Also, in the medium term, future work consists of addressing the following three topics in the adaptation logic processes:

a) *Distributed adaptation logic.* The value of separating the deployment of the affect recognition logic from the adaptation and functional logics was proven, since developers must trade affective adaptive capabilities off against functional quality attributes (such as system performance). For instance, running affect recognition and adaptation mechanisms on the same computer with the Pac-Man video game was not problematic; however, running affect recognition and adaptation mechanisms on the same computer with a game created using sophisticated 3D game engines was shown to be risky in some instances. Moving affect measurements to a second computer was shown to be feasible even though this was susceptible to network issues. However, the application of the same principle to separate the deployment of the adaptation logic and functional logic, connecting them using a networking connection, was neither considered nor evaluated.

b) *Multi-users.* The case studies run considered the implementation of systems that adapt to the affective states of one user, to which was attached one or more sensors. Sensors report to agents, agents to their central control unit, which reports to the adaptation logic. Having a system adapt to the combined affective states of more than one user implies attaching one or more sensors to each user, running their respective agents and one central control unit for each user to combine the measures of the agents handling the sensors attached to that user. Running several copies of agents and central control units is proved to be possible. However, tuning needs to be done to specify mechanisms that allow the configuration of which central control unit each agent is connected to. Similarly, tuning is needed for an adaptation logic to include more than one gauge-point, each of which is to be connected to a central control unit.

A central control unit, with their corresponding agents, is intended to handle multimodal affect recognition, informed by a group of sensors attached to one user. Having one adaptation logic connected to multiple central control units can trigger an adaptive reaction, driven by multiple users signals. While it has not been tested we see no conceptual barrier or limitation to ADASPL been rapidly tuned for systems with multiple users. However, this remains to be confirmed, through actual experimentation and implementation.

c) *Handling adaptation rules.* The ADAS framework provides two approaches to handling adaptation rules: (1) interfaces to implement adaptation rules as a method with condition-action pairs; and (2) interfaces to include the Jess rule engine. Future work can explore a third option: creating an interpreter for an ad-hoc defined domain-specific language able to facilitate, among others, the representation of affect quantification and the prior history of outcomes. A language that can combine the imperative specification of procedural algorithms and the declarative specification of decisions. Such a type of language has been already applied to others domains of adaptation (Cheng & Garlan, 2012), for instance, being able to ask “user is in a pleasured mood then…” instead of a conditional “(pleasure >= 0.5)” or having to create a list of pleasured emotions “(state == excited || state == engaged || state == meditation ...)”; similarly, it allows the creation of expressions such as “user is bored but user was engaged in the activity before the current one then…”.

Furthermore, it is important that the ADAS infrastructure incorporates the storage of the collected data, both affect measurements and system status, which is of interest to researchers for data mining and also for machine learning for model definition and testing. To accomplish this, data handling approaches need to be evaluated and best practices considered.

Long-term research beyond ADASPL could focus on addressing:

a) Security and privacy issues, given the sensitivity of the collected information, and tools and interfaces for rule definition and validation.
b) Inclusion of affect in the software engineering process to: elicit, analyze, specify, and validate affect requirements; specify and conduct an affect testing processes; analyze and perform affect maintenance; establish and maintain an affect configuration management; and handle other affect-specific parts of the process.
c) Inclusion of adaptation in the engineering process to: elicit, analyze, specify, and validate adaptation requirements; specify and conduct an adaptation testing processes; analyze and perform adaptation maintenance; establish and maintain an adaptation configuration management; and handle other adaptation-specific parts of the process.

REFERENCES

Alrajeh, D., Kramer, J., Russo, A., & Uchitel, S. (2009). Learning operational requirements from goal models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. Washington, DC, USA. (pp. 265–275). IEEE Computer Society. http://doi.org/10.1109/ICSE.2009.5070527

Arroyo, I., Cooper, D. G., Burleson, W., Woolf, B. P., Muldner, K., & Christopherson, R. M. (2009). Emotion sensors go to school. In *Proceedings of the 2009 conference on Artificial Intelligence in Education: Building Learning Systems that Care: From Knowledge Representation to Affective Modeling (AIED)*. Amsterdam, Netherlands, Netherlands. (pp. 17–24). IOS Press.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). Boston, MA, USA: Addison-Wesley.

Bickmore, T. W., & Picard, R. W. (2005). Establishing and maintaining long-term human-computer relationships. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *12*(2), 293–327. http://doi.org/10.1145/1067860.1067867

Burleson, W., & Picard, R. W. (2007). Gender-specific approaches to developing emotionally intelligent learning companions. *IEEE Intelligent Systems and Their Applications*, *22*(4), 62–69. http://doi.org/http://doi.ieeecomputersociety.org/10.1109/MIS.2007.69

Burleson, W., Picard, R. W., Perlin, K., & Lippincott, J. (2004). A platform for affective agent research. In *Proceedings of the 3rd International Conference on Autonomous Agents and Multi Agent Systems (AAMS)*. New York, NY, USA. ACM.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007). *Pattern-oriented software architecture volume 5: On patterns and pattern languages*. Wiley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture volume 1: A system of patterns* (1st ed.). Wiley.

Callele, D., Neufeld, E., & Schneider, K. (2006). Emotional requirements in video games. In *Proceedings of the 14th IEEE International Conference Requirements Engineering (RE)*. Minneapolis, Minnesota, USA. (pp. 299–302). IEEE Computer Society. http://doi.org/10.1109/RE.2006.19

Calvo, R. A., & D'Mello, S. K. (2010). Affect detection: An interdisciplinary review of models, methods, and their applications. *IEEE Transactions on Affective Computing*, *1*(1), 18–37. http://doi.org/10.1109/T-AFFC.2010.1

Camara, J., Moreno, G., & Garlan, D. (2015). Reasoning about human participation in self-adaptive systems. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Firenze, Italy. (pp. 146–156). IEEE Computer Society. http://doi.org/10.1109/SEAMS.2015.14

Cetina, C., Fons, J., & Pelechano, V. (2008). Applying software product lines to build autonomic pervasive systems. In *Proceedings of the 12th International Conference on Software Product Line (SPLC)*. Limerick, Ireland. (pp. 117–126). http://doi.org/10.1109/SPLC.2008.13

Cheng, B. H., Lemos, R., Giese, H., Inverardi, P., & Magee, J. (2009). Software engineering for self-adaptive systems: A research roadmap. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, & J. Magee (Eds.), *Software engineering for self-adaptive systems* (1st ed., pp. 1–26). Berlin, Germany. Springer. http://doi.org/10.1007/978-3-642-02161-9_1

Cheng, S. W., & Garlan, D. (2012). Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software, 85*(12). http://doi.org/10.1016/j.jss.2012.02.060

Cheng, S. W., Garlan, D., & Schmerl, B. (2005). Making self-adaptation an engineering reality. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, & S. Leonardi (Eds.), *Self-star properties in complex information systems, conceptual and practical foundations* (pp. 158–173). Berlin, Germany. Springer.

Cheng, S. W., Garlan, D., & Schmerl, B. (2009). RAIDE for engineering architecture-based self-adaptive systems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. Vancouver, British Columbia, Canada (pp. 435–436). IEEE Computer Society. http://doi.org/10.1109/icse-companion.2009.5071049

Chi, M., VanLehn, K., & Litman, D. (2010). Do micro-level tutorial decisions matter: Applying reinforcement learning to induce pedagogical tutorial tactics. In *Proceeding of the 2010 International Conference on Intelligent Tutoring Systems (ITS)*. Pittsburgh, PA, USA. (pp. 224–233). Springer.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. http://doi.org/10.1109/32.295895

Clay, A., Couture, N., & Nigay, L. (2009). Engineering affective computing: A unifying software architecture. In *Proceedings of the 3rd International Conference on Affective Computing and Intelligent Interaction (ACII)*. Amsterdam, Netherlands (pp. 1–6). IEEE. http://doi.org/10.1109/ACII.2009.5349541

Clements, P., & Northrop, L. M. (2002). *Software product lines: Practices and patterns*. Addison-Wesley.

Clements, P., Kazman, R., & Klein, M. (2001). *Evaluating software architectures: Methods and case studies* (1st ed.). Addison-Wesley.

Cooper, D. G., Arroyo, I., Woolf, B. P., Muldner, K., Burleson, W., & Christopherson, R. M. (2009). Sensors model student self-concept in the classroom. In *Proceedings of the 17th International Conference on User Modeling, Adaptation, and Personalization (UMAP)*. Trento, Italy. (pp. 30–41). Springer. http://doi.org/10.1007/978-3-642-02247-0_6

Crnkovic, I. (2001). Component-based software engineering – new challenges in software development. *Software Focus*, *2*(4), 127–133.

Deugo, D. (1998). Foundation patterns. In *Proceedings of the 5th Pattern Languages of Programs Conference (PLOP)*.

Ekman, P. (1992). Are there basic emotions? *Psychological Review*, *99*(3), 550–553.

Fairbanks, G. H. (2010). *Just enough software architecture: A risk-driven approach* (1st ed.). Marshall & Brainerd.

Fant, J., Gomaa, H., & Pettit, R. (2012). Software product line engineering of space flight software. In *Proceedings of the 3rd International Workshop on Product Line Approaches in Software Engineering (PLEASE)*. Zurich, Switzerland. (pp. 41–44). IEEE. http://doi.org/10.1109/PLEASE.2012.6229769

Feldman, S. (2004). A conversation with Alan Kay. *Queue, 2*(9), 20.

Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, A. B., … & Krikava, F. (2015). Software engineering meets control theory. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Firenze, Italy. (pp. 71-82). IEEE. http://doi.org/10.1109/SEAMS.2015.12

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software* (1st ed.). Boston, MA, USA. Addison-Wesley.

Garlan, D., & Shaw, M. (1994). *An introduction to software architecture*. Pittsburgh, PA, USA. Carnegie Mellon University.

Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., & Steenkiste, P. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, *37*(10), 46–54. http://doi.org/10.1109/MC.2004.175

Gilroy, S. W., Cavazza, M., & Benayoun, M. (2009). Using affective trajectories to describe states of flow in interactive art. In *Proceedings of the International Conference on Advances in Computer Entertainment Technology (ACE)*. Athens, Greece. (pp. 165–172). ACM.

Gomaa, H. (2004). *Designing software product lines with UML: From use cases to pattern-based software architectures*. Addison-Wesley Professional.

Gonzalez-Sanchez, J., Chavez-Echeagaray, M. E., Atkinson, R., & Burleson, W. (2011a). ABE: An agent-based software architecture for a multimodal emotion recognition framework. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Boulder, CO, USA. (pp. 187–193). IEEE. http://doi.org/10.1109/WICSA.2011.32

Gonzalez-Sanchez, J., Chavez-Echeagaray, M. E., Atkinson, R., & Burleson, W. (2011b). Affective computing meets design patterns: A pattern-based model for a multimodal emotion recognition framework. In *Proceedings of the 16th European Conference on Pattern Languages of Programs (EUROPLOP)*. Bavaria, Germany. (pp. 14:1–14:11). ACM.

Gonzalez-Sanchez, J., Chavez-Echeagaray, M. E., Atkinson, R., & Burleson, W. (2012). Towards a pattern language for affective systems. In *Proceedings of the 19th Conference on Pattern Languages of Programs (PLOP)*, Tucson, AZ, USA. (pp. 1–23). ACM.

Gonzalez-Sanchez, J., Chavez-Echeagaray, M. E., Gibson, D., & Atkinson, R. (2013). Multimodal affect recognition in virtual worlds: Avatars mirroring user's affect. In *Proceedings of the 2013 HUMAINE Association Conference on Affective Computing and Intelligent Interaction (ACII)*. Geneva, Switzerland. (pp. 724-725). IEEE Computer Society.

Gonzalez-Sanchez, J., Chavez-Echeagaray, M. E., VanLehn, K., Burleson, W., Girard, S., Hidalgo-Pontet, Y., & Zhang, L. (2014). A system architecture for affective meta intelligent tutoring systems. In *Proceeding of the 12th International Conference on Intelligent Tutoring Systems (ITS)*. Honolulu Hawaii, USA. (pp. 529–534). Springer.

Gonzalez-Sanchez, J., Christopherson, R. M., Chavez-Echeagaray, M. E., Gibson, D. C., Atkinson, R., & Burleson, W. (2011c). How to do multimodal detection of affective states? In *Proceedings of the 11th IEEE International Conference on Advanced Learning Technologies (ICALT*). Athens, GA, USA. (pp. 654–655). IEEE. http://doi.org/10.1109/ICALT.2011.206

Gowri, R., Kanmani, S., Induja, R., Hemalatha, A., & Devi, M. (2010). A secure agent based intelligent tutoring system using FRS. In *Proceedings of the 3rd International Conference on Emerging Trends in Engineering and Technology (ICETET)*. Goa, India. (pp. 5–10). IEEE Computer Society. http://doi.org/10.1109/ICETET.2010.135

Greenfield, J., & Short, K. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In *Proceedings of the 18th International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Anaheim, CA, USA. (pp. 16–27). ACM. http://doi.org/10.1145/949344.949348

Hallsteinsen, S., Hinchey, M., Park, S., & Schmid, K. (2008). Dynamic software product lines. *IEEE Computer*, *41*(4), 93–95. http://doi.org/10.1109/MC.2008.123

Hong, J.Y., Suh, E.H., & Kim, S.J. (2009). Context-aware systems: A literature review and classification. *Expert Systems with Applications, 36*(4) 8509–8522. http://doi.org/10.1016/j.eswa.2008.10.071

Hussain, M. S., & Calvo, R. A. (2009). *A framework for multimodal affect recognition*. Learning System Group, DECE, University of Sydney.

Jacob, B., Lanyon-Hogg, R., Nadgir, D., & Yassin, A. (2004). *A practical guide to the IBM autonomic computing toolkit.* IBM Redbooks.

Jaimes, A., & Sebe, N. (2007). Multimodal human-computer interaction: A survey. *Computer Vision and Image Understanding*, *108*(1-2), 116–134.

Janisse, M. P. (1973). Pupil size and affect: A critical review of the literature since 1960. *Canadian Psychologist*, *14*(4), 311–329. http://doi.org/10.1037/h0082230

Johnson, R. E. (1992). Documenting frameworks using patterns. In *Proceedings of the 11st International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. Vancouver, B.C., Canada. (pp. 63–76). ACM. http://doi.org/10.1145/141936.141943

Johnson, R. E. (1997). Components, frameworks, patterns. *ACM SIGSOFT Software Engineering Notes*, *22*(3), 10–17. http://doi.org/10.1145/258368.258378

Kaliouby, El, R., & Robinson, P. (2005). Generalization of a vision-based computational model of mind-reading. In *Proceedings of the 3rd International Conference on Affective Computing and Intelligent Interaction (ACII)*. Amsterdam, Netherlands. (pp. 582–589). Springer.

Kapoor, A., & Picard, R. W. (2005). Multimodal affect recognition in learning environments. In *Proceedings of the 13th International Conference on Multimedia (MULTIMEDIA)*. Melbourne, VIC, Australia. (pp. 677–682). ACM. http://doi.org/10.1145/1101149.1101300

Kapoor, A., Burleson, W., & Picard, R. W. (2007). Automatic prediction of frustration. *International Journal of Human-Computer Studies*, *65*(8), 724–736.

Kastner, C., Apel, S., & Kuhlemann, M. (2008). Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. Leipzig, Germany. (pp. 311–320). ACM. http://doi.org/10.1145/1368088.1368131

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, *36*(1), 41–50. http://doi.org/10.1109/MC.2003.1160055

Koelstra, S., Muhl, C., Soleymani, M., Lee, J.-S., Yazdani, A., Ebrahimi, T., … Patras, I. (2012). DEAP: A database for emotion analysis; Using physiological signals. *IEEE Transactions on Affective Computing*, *3*(1), 18–31. http://doi.org/10.1109/T-AFFC.2011.15

Kruchten, P. (2004). *The Rational unified process: An introduction* (3rd ed.). Addison-Wesley.

Lehman, B., D'Mello, S. K., & Person, N. (2008). All alone with your emotions: An analysis of student emotions during effortful problem solving activities. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems (ITS)*. Montreal, Canada. Springer.

Lewandowsky, S., Oberauer, K., Yang, L. X., & Ecker, U. K. (2010). A working memory test battery for MATLAB. *Behavior Research Methods*, *42*(2), 571–585.

Lisetti, C. L., & Nasoz, F. (2002). MAUI: A multimodal affective user interface. In *Proceedings of the 10th International Conference on Multimedia (MULTIMEDIA)*. Juan-les-Pins, France. (pp. 161–170). ACM. http://doi.org/10.1145/641007.641038

Maat, L., & Pantic, M. (2007). Gaze-X: Adaptive, affective, multimodal interface for single-user office scenarios. In *Proceedings of the 2007 International Conference on Artificial Intelligence for Human Computing (ICAIHC)*. (pp. 251–271). Springer.

Martin, R. C. (2013). *Agile software development, principles, patterns, and practices*. Pearson.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *2*(4), 308–320. http://doi.org/10.1109/TSE.1976.233837

Mehrabian, A. (1996). Pleasure-arousal-dominance: A general framework for describing and measuring individual differences in temperament. *Current Psychology*, *14*(4), 261–292.

Morais, Y., Burity, T., & Elias, G. (2009). A systematic review of software product lines applied to mobile middleware. In *Proceedings of the 6th International Conference on Information Technology: New Generations (ITNG)*. Las Vegas, Nevada, USA. (pp. 1024–1029). IEEE Computer Society. http://doi.org/10.1109/ITNG.2009.82

Mota, S., & Picard, R. W. (2003). Automated posture analysis for detecting learner's interest level. In *Proceedings of the 2003 Workshop on Computer Vision and Pattern Recognition (CVPRW)*. (Vol. 5, pp. 49–55). IEEE.

Nkambou, R. (2006). A framework for affective intelligent tutoring systems. In *Proceedings of the 7th International Conference on Information Technology Based Higher Education and Training (ITHET).* IEEE. http://doi.org/10.1109/ITHET.2006.339720

Norman, D. A., Ortony, A., & Russell, D. M. (2003). Affect and machine design: Lessons for the development of autonomous machines. *IBM Systems Journal*, *42*(1), 38–44.

Norvig, P., & Cohen, D. (1997). Adaptive software. *PC AI*, *11*(1), 27-30. Retrieved from http://norvig.com/adapaper-pcai.html

Picard, R. W. (1997). *Affective computing* (1st ed.). Cambridge, MA, USA. MIT Press.

Picard, R. W. (2010). Affective computing: From laughter to IEEE. *IEEE Transactions on Affective Computing*, *1*(1), 11–17. http://doi.org/10.1109/T-AFFC.2010.10

Possompes, T., Dony, C., Huchard, M., Rey, H., Tibermacine, C., & Vasques, X. (2010). Towards software product lines application in the context of a smart building project. In *Proceedings of the 2nd International Workshop on Model-driven Product Line Engineering (MDPLE)*. France. (pp. 73–84).

Qi, Y., & Picard, R. W. (2002). Context-sensitive Bayesian classifiers and application to mouse pressure pattern classification. In *Proceedings of the 16th International Conference on Pattern Recognition (ICPR)*. Quebec City, QC, Canada**.** (Vol. 3, pp. 448–451). IEEE. http://doi.org/10.1109/ICPR.2002.1047973

Ramirez, A. J., & Cheng, B. H. (2010). Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Cape Town, South Africa. (pp. 49–58). ACM. http://doi.org/10.1145/1808984.1808990

Reynolds, C., & Picard, R. W. (2004). Affective sensors, privacy, and ethical contracts. In *Extended Abstracts on Human Factors in Computing Systems (CHI EA*). Vienna, Austria. (pp. 1103–1106). ACM. http://doi.org/10.1145/985921.985999

Rudolf, G. (2003). *Some guidelines for deciding whether to use a rules engine.* Sandia National Labs. Retrieved from http://herzberg.ca.sandia.gov/guidelines.shtml

Russell, J. (1980). A circumplex model of affect. *Journal of Personality and Social Psychology*, *39*(6), 1161–1178.

Schmidt, D. C., & Buschmann, F. (2003). Patterns, frameworks, and middleware: Their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. Portland, Oregon, USA. (pp. 694–704). IEEE.

Schroder, M. (2010). The SEMAINE API: Towards a standards-based framework for building emotion-oriented systems. *Journal of Advances in Human-Computer Interaction*, *2010*. http://doi.org/10.1155/2010/319406

Sebe, N., Cohen, I., & Huang, T. S. (2005). Multimodal emotion recognition. *Handbook of Pattern Recognition and Computer Vision,* 4, 387-419.

Shaw, M. (1995). Beyond objects: A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, *20*(1), 27–38. http://doi.org/10.1145/225907.225911

Shen, L., Peng, X., & Zhao, W. (2012). Software product line engineering for developing self-adaptive systems: Towards the domain requirements. In *Proceedings of the 36th Conference on Computer Software and Applications (COMPSAC)*. Izmir, Turkey. (pp. 289–296). IEEE. http://doi.org/10.1109/COMPSAC.2012.40

Shute, V., & Zapata-Rivera, D. (2007). Adaptive technologies. In D. H. Jonassen & M. J. Spector (Eds.), *Handbook of research on educational communications and technology.* (pp. 277–294). Princeton, NJ, USA. Educational Testing Service.

Siegmund, N., Pukall, M., Soffner, M., Koppen, V., & Saake, G. (2009). Using software product lines for runtime interoperability. In *Proceedings of the 6th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*. Genova, Italy. (pp. 4:1–4:7). ACM. http://doi.org/10.1145/1562860.1562864

Sommerville, I. (2002). *Software engineering*. Addison-Wesley.

Stavru, S. (2011). Managing change in self-adaptive software systems. In *Proceedings of the 12th International Conference on Computer Systems and Technologies (COMPSYSTECH)*. Palermo, Italy. (pp. 647–652). ACM. http://doi.org/10.1145/2023607.2023715

Strauss, M., Reynolds, C., Hughes, S., Park, K., McDarby, G., & Picard, R. W. (2005). The handwave bluetooth skin conductance sensor. In *Proceedings of the 2005 International Conference on Affective Computing and Intelligent Interaction (ACII)*. Beijing, China. (pp. 699–706). Springer.

Tomkins, S. (1962). Affect imagery consciousness (Vols. 1–3).

Vallabhaneni, A., Wang, T., & He, B. (2005). Brain-computer interface. *Neural Engineering*, 85-121. Boston, MA, USA. Springer. http://doi.org/10.1007/0-306-48610-5_3

Van den Broek, E. L. (2011). Ubiquitous emotion-aware computing. *Personal and Ubiquitous Computing*, *17*(1), 53–67. http://doi.org/10.1007/s00779-011-0479-9

VanLehn, K., Burleson, W., Girard, S., Chavez-Echeagaray, M. E., Gonzalez-Sanchez, J., Hidalgo-Pontet, Y., & Zhang, L. (2014). The affective meta-tutoring project: Lessons learned. In *Proceeding of the 12th International Conference on Intelligent Tutoring Systems (ITS)*. Honolulu, Hawaii. (pp. 84–93). Springer.

Vildjiounaite, E., Kyllonen, V., Vuorinen, O., Makela, S. M., Keranen, T., Niiranen, M., … Peltola, J. (2009). Requirements and software framework for adaptive multimodal affect recognition. In *Proceedings of the 3rd International Conference on Affective Computing and Intelligent Interaction (ACII)*. Amsterdam, The Netherlands. IEEE. http://doi.org/10.1109/ACII.2009.5349393

Wagner, J., André, E., & Jung, F. (2009). Smart sensor integration: A framework for multimodal emotion recognition in real-time. In *Proceedings of the 3rd International Conference on Affective Computing and Intelligent Interaction (ACII).* Amsterdam, The Netherlands. IEEE. http://doi.org/10.1109/ACII.2009.5349571

Weiss, D. M., & Lai, C. T. R. (1999). *Software product-line engineering* (1st ed.). Addison-Wesley.

Weyns, D., Malek, S., & Andersson, J. (2010). On decentralized self-adaptation: lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Cape Town, South Africa. (pp. 84–93). ACM. http://doi.org/10.1145/1808984.1808994

Zhang, L., VanLehn, K., Girard, S., Burleson, W., Chavez-Echeagaray, M. E., Gonzalez-Sanchez, J., & Hidalgo-Pontet, Y. (2014). Evaluation of a meta-tutor for constructing models of dynamic systems. *Computers & Education*, *75*(C), 196–217. http://doi.org/10.1016/j.compedu.2014.02.015

APPENDIX A
INSTALLATION OF THE ADAS FRAMEWORK

The guidelines assume that the ADAS framework is installed and ready to be used. To do so, copy the ADAS distribution to the host computer. The ADAS distribution includes the following files:

```
adas.jar
lib/jna-4.1.0.jar
lib/opencsv-1.8.jar
lib/RXTXcomm.jar
lib/rxtxSerial.dll
```

The full infrastructure is contained in a JAR file and a lid folder contains libraries required by the infrastructure as follows:

a) JNA, acronym for Java Native Access, is a library that provides Java programs easy access to native shared libraries, such as the libraries (DLL files) of Emotiv and Tobi SDKs.
b) openCSV is a CSV (comma-separated values) parser library for Java.
c) RXTXcomm is a Java library that in conjunction with the native library rxtx.dll provides serial and parallel communication for Java in Windows platforms.

Additionally, to use Emotiv headset the following dll files from Emotiv SDK should be available in the system path or added to the lib folder

```
lib/edk.dll
```

```
   lib/edk_utils.dll
```

And, in order to run the example with Jess rule systems and/or use Jess for complex adaptation handling, Jess library should be added in the lib folder also

```
   lib/jess.dll
```

Developers and users can execute adas.jar, double click or using java command line, to access ASAS application tools. Running the adas.jar will open a dialog box, as showing in Figure A.1., asking for the application that they want to run. The options include:

a) six pre-build agents for commercial Emotiv headset and Tobii eye tracker, for MindReader software, as well as, for in-house built skin conductance, pressure, and posture sensors. They are links to execute instances of applications that reside in the `adas.gui.agent` package in `Apanel` subclasses: `APanelEmotiv, APanelTobii, APanelMR, ApanelSkin, APanelPressure,` and `APanelPosture.`
b) a prebuilt agent that does not require to connect to any physical device but generate random values and It can be used for application testing purposes. It is a link to execute `adas.gui.agent.APanelDummy.`
c) the central control unit dashboard, a link to execute `adas.gui.centre.CentreDashboard`, and

d) the general ADAS dashboard, a link to execute `adas.gui.main.Dashboard`.

Developers can include adas.jar and the libraries in their project to take advantage of the framework. The guidelines to accomplish the scenarios described in Chapter 3 are described in the following sections.

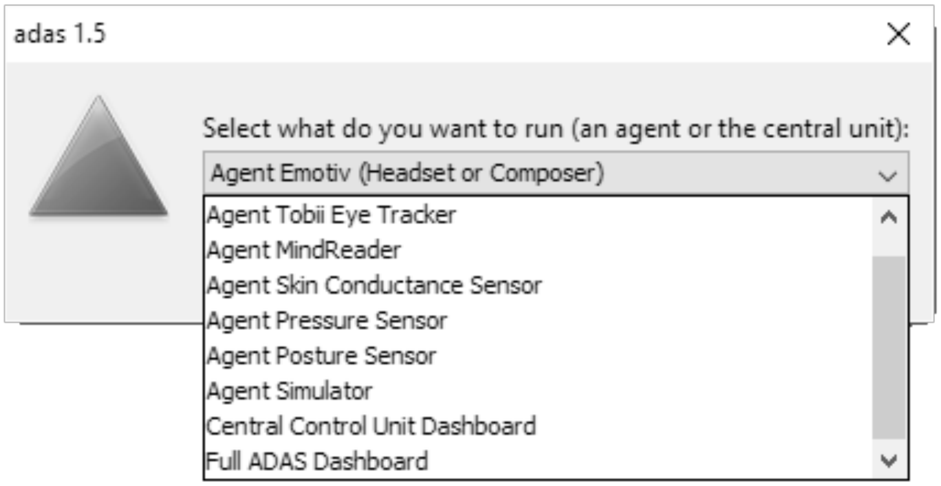

adas 1.5
Select what do you want to run (an agent or the central unit):
Agent Emotiv (Headset or Composer)
Agent Tobii Eye Tracker
Agent MindReader
Agent Skin Conductance Sensor
Agent Pressure Sensor
Agent Posture Sensor
Agent Simulator
Central Control Unit Dashboard
Full ADAS Dashboard

Figure A.1. Dialog box after the execution of ADAS.

APPENDIX B
CAPSTONE FACULTY PROJECT PROPOSAL

School of Computing and Informatics Decision Systems Engineering Computer Science and Engineering Capstone Faculty Project Proposal

1. Contact Information

Faculty Contact Information

Name	**Dr. Robert Atkinson**	**ASU Email**	**Robert.Atkinson@asu.edu**

Project Contact Information (if different from Proposer)

Name	**Javier Gonzalez-Sanchez**	**ASU Email**	**javiergs@asu.edu**

2. Project Description

a. Project title:

Affect-Driven Adaptive Systems:
Creating software systems able to adapt itself to user's emotions and interest.

b. Project description:

Affect (experiencing a feeling or emotion) shapes our vision of the world and how we experience it, furthermore, affect influences our rational decision-making process and action selection. That awareness has created a trend toward **improving technology** by providing it with the intelligent ability to simulate empathy by **recognizing human affect** and **adapting its behavior** driven by that affect.

We look forward to leverage the inclusion of affect-driven adaptation capabilities in an environment through inspiring and supporting student creativity to develop an affective adaptive system of their own inspiration. We aim to provide teams with the tools (hardware and software) that support emotion and interest recognition and mentoring, while challenge them to imagine, design, and build a system that exploits those capabilities such as, but not limited to, a next-generation learning environment, a more engaging video-game, an empathetic health support system.

We will support the creation of system which functionality includes:

- Collecting affect signals from the user through a set of diverse sensing devices including but not limited to wireless brain computer interfaces, mobile eye tracking, face-based gesture recognition, and wearable physiological sensors. We are able to provide all of them for students to play and teach students about how to use them.

- Combine the affect signals with information about the events of the user's interaction with the environment, i.e., relate the user feelings with what he/she is doing and what are the results she/he is getting from the system.

- Define the behaviors to be implemented to improve user's experience.

Examples of products that have been developed in previous semesters by fellow capstone teams include:

- **Lost in the Dark**. An adaptable 3D video game, which uses affect measurements as input to alter and adjust the gaming environment; brainwaves were used as input to infer meditation, excitement, and engagement; lighting and colors were altered in the game according to the user's affective state.

  Bernays, R., Mone, J., Yau, P., Murcia, M., Gonzalez-Sanchez, J., Chavez-Echeagaray, M. E., Christopherson, R. M., Atkinson, R., and Yoshihiro, K. (2012). Lost in the Dark: Emotion Adaption. In Adjunct proceedings of the 25th annual ACM symposium on User interface software and technology (UIST 2012). ACM.

- **Empathetic Second Life® Avatars.** A generic real-time multimodal affect recognition hub was integrated within an online virtual world to make the virtual world's avatar mirror its owner's affect aiming to increase believability, likability, trustability, and enjoyability, as well as to create long-lasting relationships.

  Gonzalez-Sanchez, J., Chavez-Echeagaray, M. E., Gibson, D., and Atkinson, R. (2013). Multimodal Affect Recognition in Virtual Worlds: Avatars Mirroring User's Affect. In Proceedings of the 2013 Humaine Association Conference on Affective Computing and Intelligent Interaction (ACII 2013). IEEE Computer Society.

- **Including Affect-Driven Adaptation to the Pac-Man Video Game.** The well-known game Pac-Man was modified to provide affect-driven adaptive capabilities; the game changes aiming to keep or increase player's engagement.

Harris, A., Hoch, A., Kral, R., Teposte, M., Villa, A., Chavez-Echeagaray, M.E., Gonzalez-Sanchez, J., and Atkinson, R. (2014). Including Affect-driven Adaptation to the Pac-Man Video Game. In Extended Abstracts Proceedings of the 18th International Symposium on Wearable Computers (ISCW 2014). ACM. In press.

The tools available on our lab that students could use include:

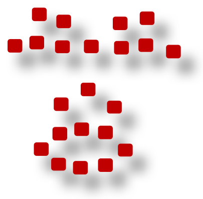
- Brain computer interfaces, such as X10 and X24 ABM headset and Emotiv headset.

X10 and X24 ABM headset	Emotiv headset

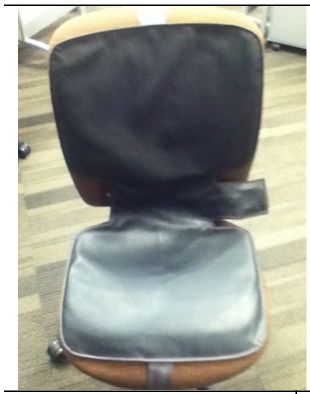
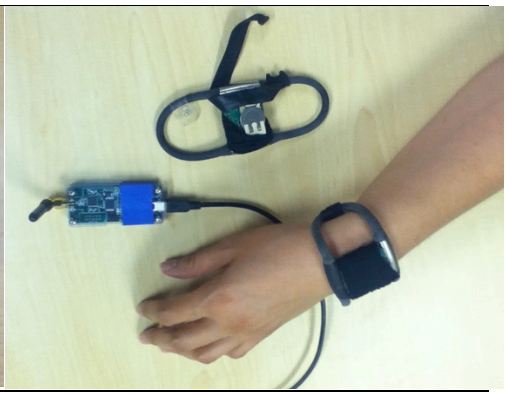
- Eye tracking systems,

Tobii glasses eye tracking system	Tobii wide screen eye tracking system

- Face-based gesture recognition

		interested agreeing concentrating disagreement thinking unsure

- Wearable physiological sensors

		
Posture sensor	Pressure sensor	Skin conductance sensor

c. Deliverables:

By the end of the year it is expected that students deliver an environment able to adapt itself driven by affect. It is expected that the students include assets such as software design, testing, validation, and documentation.
Additionally, we will be encouraging students to present their work on international conferences as a demo, poster, or short paper. If students' work is accepted to be presented, we will support them to attend (travel) to the conference.

d. Motivation:

Explode the use of affective-driven adaptive capabilities in software systems to enhance user's experience.

e. Student learning experience:

This project represents a singular real-world experience for students in several aspects, including:

- **Solving real-world problems with state-of-the-art technology.** We aim to create a singular opportunity to live a real world experience where creativity is encouraged but boundaries in the organization artifacts are defined. Students are not going to be building a trivial system from scratch; students will be applying systematic reuse of previous work to innovate in technology usages that improve human life.

- **Teamwork Experience**. Students will grow up their teamwork skills working in a project where new requirements drive the creation of new software components;

students will use their analysis, design, programming, testing, and project management skills, collaborating with their team mates but also with mentors and researchers.

- **Component-driven pattern-based development**. As a developing team, we work under a component-driven approach. Student's responsibilities goes beyond programming and documenting; it will be required that design and implementation follow the component-driven approach, use design patterns, and satisfy organization rules. The goal is to create a product but also assure that what is done will be the basis for future work. As a team, we support several efforts in learning environments (from video games to tutoring systems); therefore, our group guideline for software is the creation of families of products instead of isolated efforts.

- **Human Computer Interfaces**. Students will have the opportunity to learn more about modern human computer interfaces including but not limited to use and work with brain-computer interfaces, eye tracking systems, and physiological sensors.

- **Research experience.** Students will be collaborating with researchers and will be encouraged to get involved in research activities, such as paper writing, poster design, and conference presentation.

f. Required background:

It is expected that participant students have the following skills:

- Software development background.
- Proficient in a programming language.
- Interest in human-computer interaction and/or user-interface design will be a plus.

3. Other

Details about our work and related projects that we had supported could be seen in the following links (videos):

- Our lab research foci - http://angle.lab.asu.edu/site/
- Lost in the Dark (videogame) poster and video - http://angle.lab.asu.edu/site/?p=1330
- Affect Recognition in Virtual Worlds - http://angle.lab.asu.edu/site/?p=1746
- A Sun Devil Story in our lab - http://angle.lab.asu.edu/site/?p=1523

APPENDIX C
COMPLEXITY AND STRUCTURAL METRICS REPORTS

Report Generated by RSM

Reports were generated grouping files in alphabetical order in groups of a maximum of 20 (the limited number of files in the tool evaluation version). Reports were generated for:

a) Project C1. A total of 53 files are grouped in three partial reports with: files from AppSettings.java to Meter.java (20 files), files from MeterAnalyzer.java to SerializationManager.java (20 files), and files from SerialSensor.java to TCPServer.java (13 files), respectively. Figures C.1, C.2, and C.3 show these reports.

b) Project R1. 1 file is included in a report, as shown in Figure C.4.

c) A freeware implementation of the Pac-Man video game used as a basis to develop project C3. 11 files are included in a report, as shown in Figures C.5.

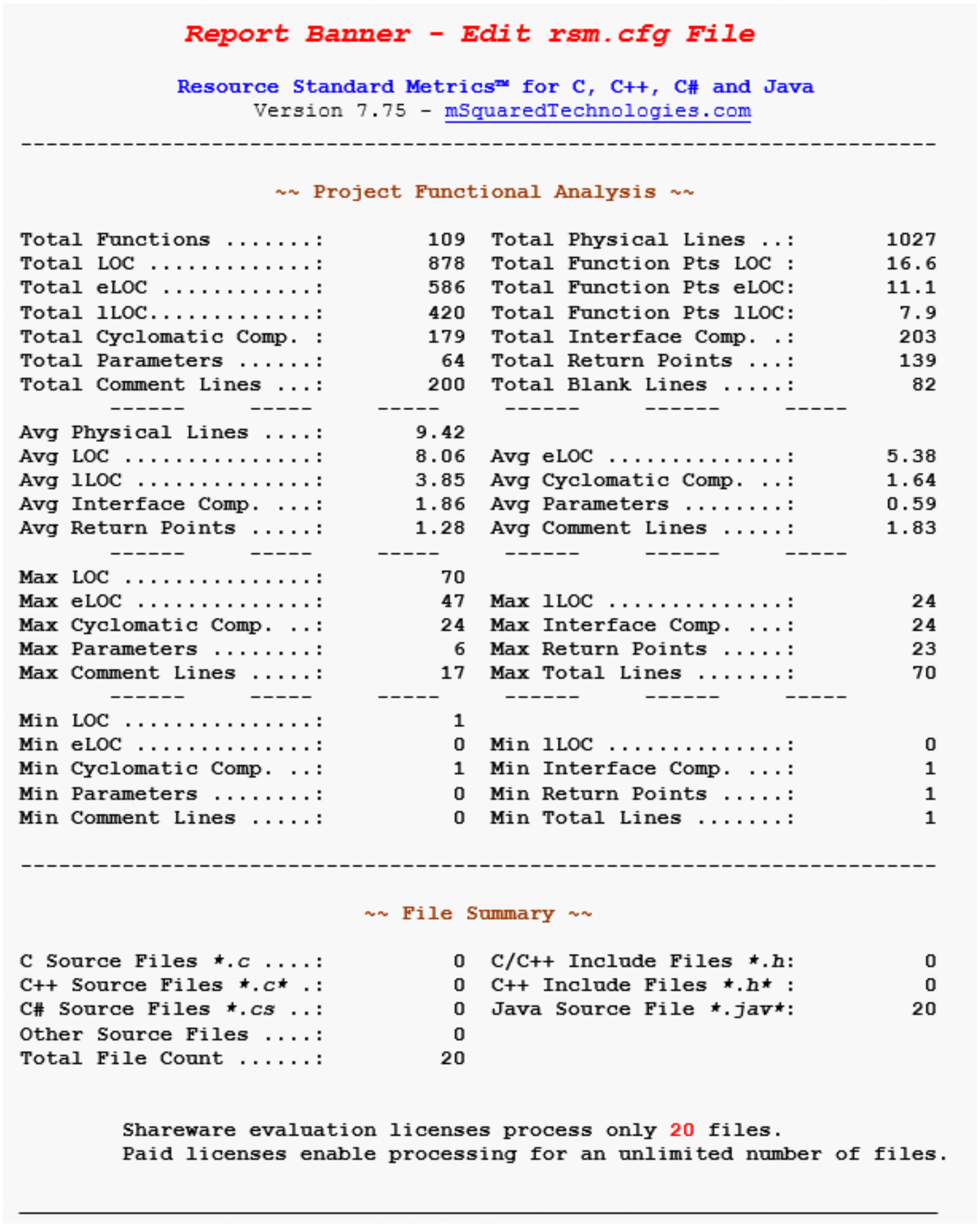
d) Project C3. 14 files are included in a report, as shown in Figure C.6.

e) Project C4. 4 files are included in a report, as shown in Figure C.7.

f) Package AC in project R2. 36 files are grouped in 2 partial reports, each with 18 files. Figures C.8 and C.9 show these reports.

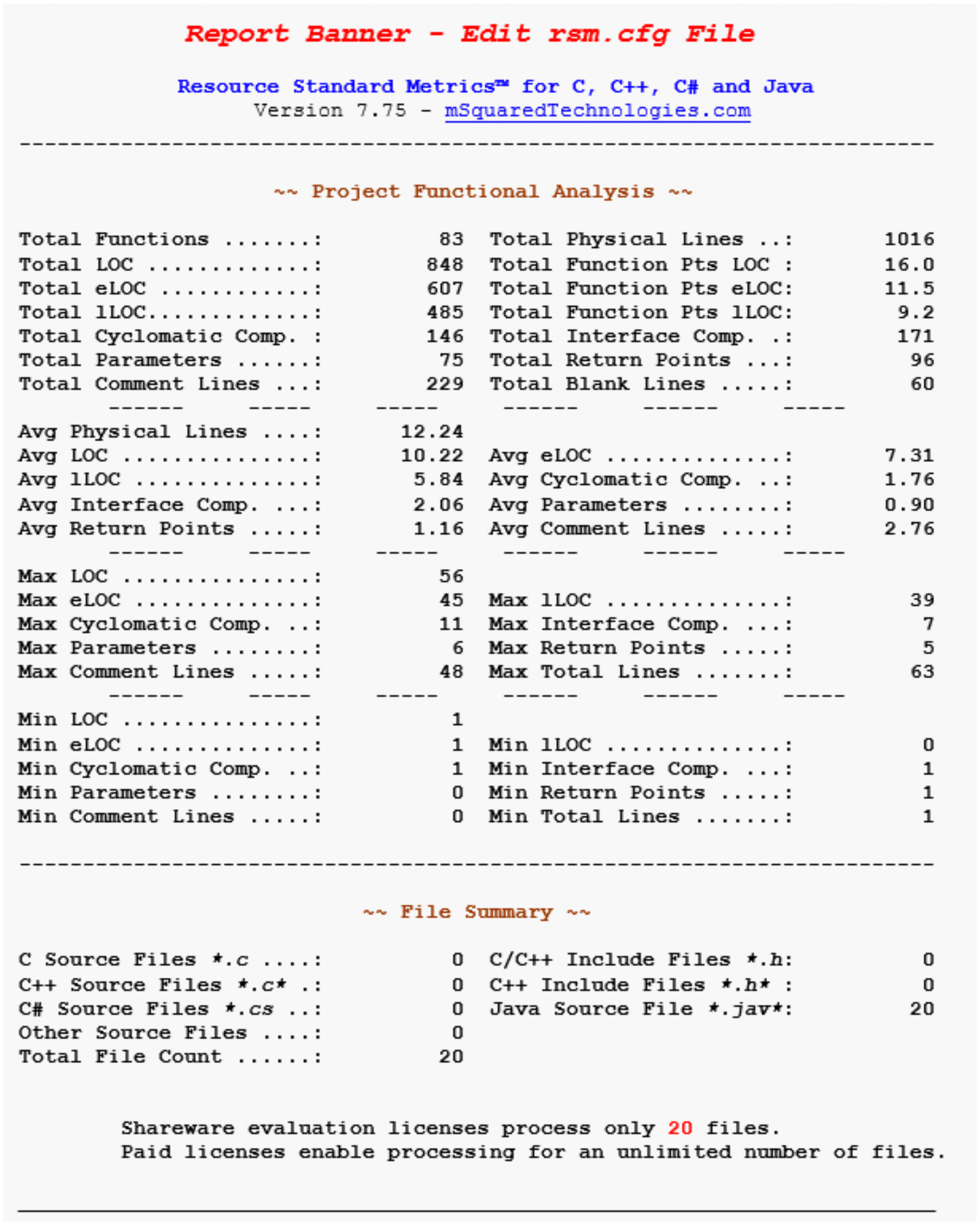
g) Legacy code used to create package AC in project R2. 8 files are included in a report, as shown in Figure C.10.

h) Package ELE (affective companion) in project R3. 4 files are included in a report, as shown in Figure C.11.



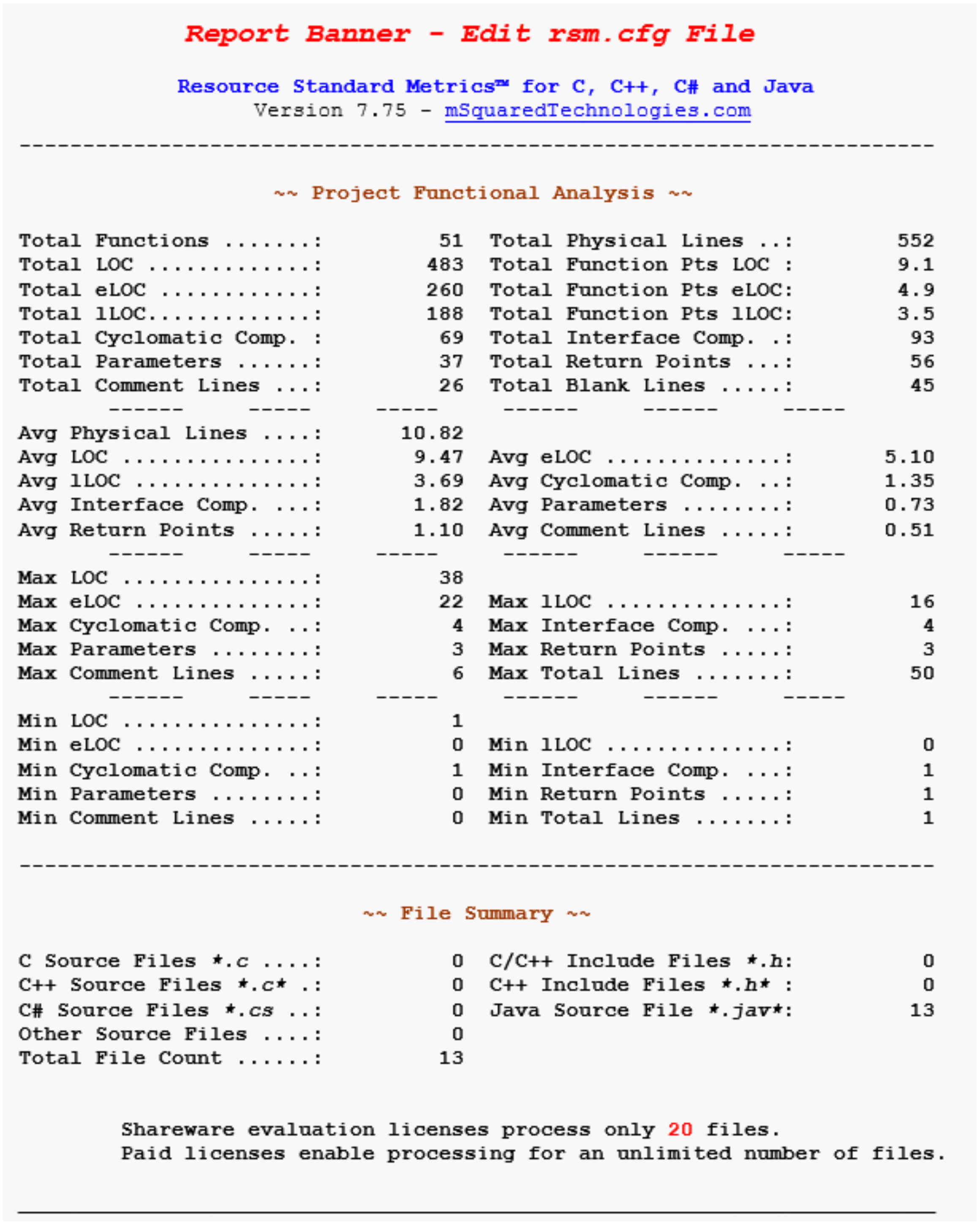
```
                 Report Banner - Edit rsm.cfg File

              Resource Standard Metrics™ for C, C++, C# and Java
                    Version 7.75 - mSquaredTechnologies.com

------------------------------------------------------------------------

                    ~~ Project Functional Analysis ~~

Total Functions .......:        109 Total Physical Lines ..:       1027
Total LOC .............:        878 Total Function Pts LOC :       16.6
Total eLOC ............:        586 Total Function Pts eLOC:       11.1
Total lLOC.............:        420 Total Function Pts lLOC:        7.9
Total Cyclomatic Comp. :        179 Total Interface Comp. .:        203
Total Parameters ......:         64 Total Return Points ...:        139
Total Comment Lines ...:        200 Total Blank Lines .....:         82
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:       9.42
Avg LOC ...............:       8.06 Avg eLOC ..............:       5.38
Avg lLOC ..............:       3.85 Avg Cyclomatic Comp. ..:       1.64
Avg Interface Comp. ...:       1.86 Avg Parameters ........:       0.59
Avg Return Points .....:       1.28 Avg Comment Lines .....:       1.83
       ------     -----     -----     ------     ------     -----
Max LOC ...............:         70
Max eLOC ..............:         47 Max lLOC ..............:         24
Max Cyclomatic Comp. ..:         24 Max Interface Comp. ...:         24
Max Parameters ........:          6 Max Return Points .....:         23
Max Comment Lines .....:         17 Max Total Lines .......:         70
       ------     -----     -----     ------     ------     -----
Min LOC ...............:          1
Min eLOC ..............:          0 Min lLOC ..............:          0
Min Cyclomatic Comp. ..:          1 Min Interface Comp. ...:          1
Min Parameters ........:          0 Min Return Points .....:          1
Min Comment Lines .....:          0 Min Total Lines .......:          1

------------------------------------------------------------------------

                           ~~ File Summary ~~

C Source Files *.c ....:          0 C/C++ Include Files *.h:          0
C++ Source Files *.c* .:          0 C++ Include Files *.h* :          0
C# Source Files *.cs ..:          0 Java Source File *.jav*:         20
Other Source Files ....:          0
Total File Count ......:         20


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.
```

Figure C.1. RSM report for project C1 (part 1 of 3) – files from AppSettings.java to Meter.java.



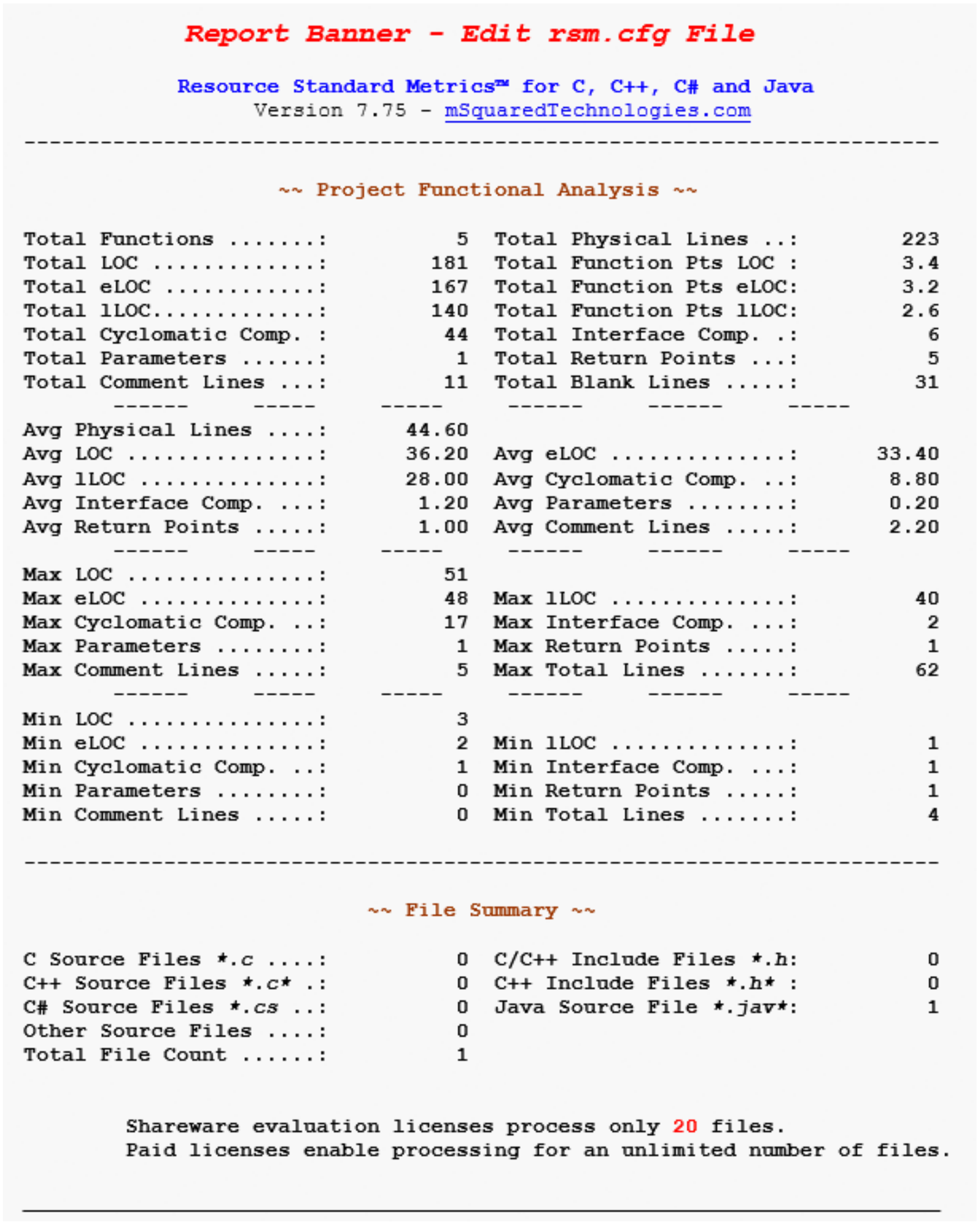
```
                Report Banner - Edit rsm.cfg File

           Resource Standard Metrics™ for C, C++, C# and Java
                 Version 7.75 - mSquaredTechnologies.com
------------------------------------------------------------------------

                 ~~ Project Functional Analysis ~~

Total Functions .......:        83  Total Physical Lines ..:      1016
Total LOC .............:       848  Total Function Pts LOC :      16.0
Total eLOC ............:       607  Total Function Pts eLOC:      11.5
Total lLOC.............:       485  Total Function Pts lLOC:       9.2
Total Cyclomatic Comp. :       146  Total Interface Comp. .:       171
Total Parameters ......:        75  Total Return Points ...:        96
Total Comment Lines ...:       229  Total Blank Lines .....:        60
       ------     -----      -----    ------     ------     -----
Avg Physical Lines ....:     12.24
Avg LOC ...............:     10.22  Avg eLOC ..............:      7.31
Avg lLOC ..............:      5.84  Avg Cyclomatic Comp. ..:      1.76
Avg Interface Comp. ...:      2.06  Avg Parameters ........:      0.90
Avg Return Points .....:      1.16  Avg Comment Lines .....:      2.76
       ------     -----      -----    ------     ------     -----
Max LOC ...............:        56
Max eLOC ..............:        45  Max lLOC ..............:        39
Max Cyclomatic Comp. ..:        11  Max Interface Comp. ...:         7
Max Parameters ........:         6  Max Return Points .....:         5
Max Comment Lines .....:        48  Max Total Lines .......:        63
       ------     -----      -----    ------     ------     -----
Min LOC ...............:         1
Min eLOC ..............:         1  Min lLOC ..............:         0
Min Cyclomatic Comp. ..:         1  Min Interface Comp. ...:         1
Min Parameters ........:         0  Min Return Points .....:         1
Min Comment Lines .....:         0  Min Total Lines .......:         1

------------------------------------------------------------------------

                         ~~ File Summary ~~

C Source Files *.c ....:         0  C/C++ Include Files *.h:         0
C++ Source Files *.c* .:         0  C++ Include Files *.h* :         0
C# Source Files *.cs ..:         0  Java Source File *.jav*:        20
Other Source Files ....:         0
Total File Count ......:        20


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.
```

Figure C.2. RSM report for project C1 (part 2 of 3) – from MeterAnalyzer.java to SerializationManager.java.



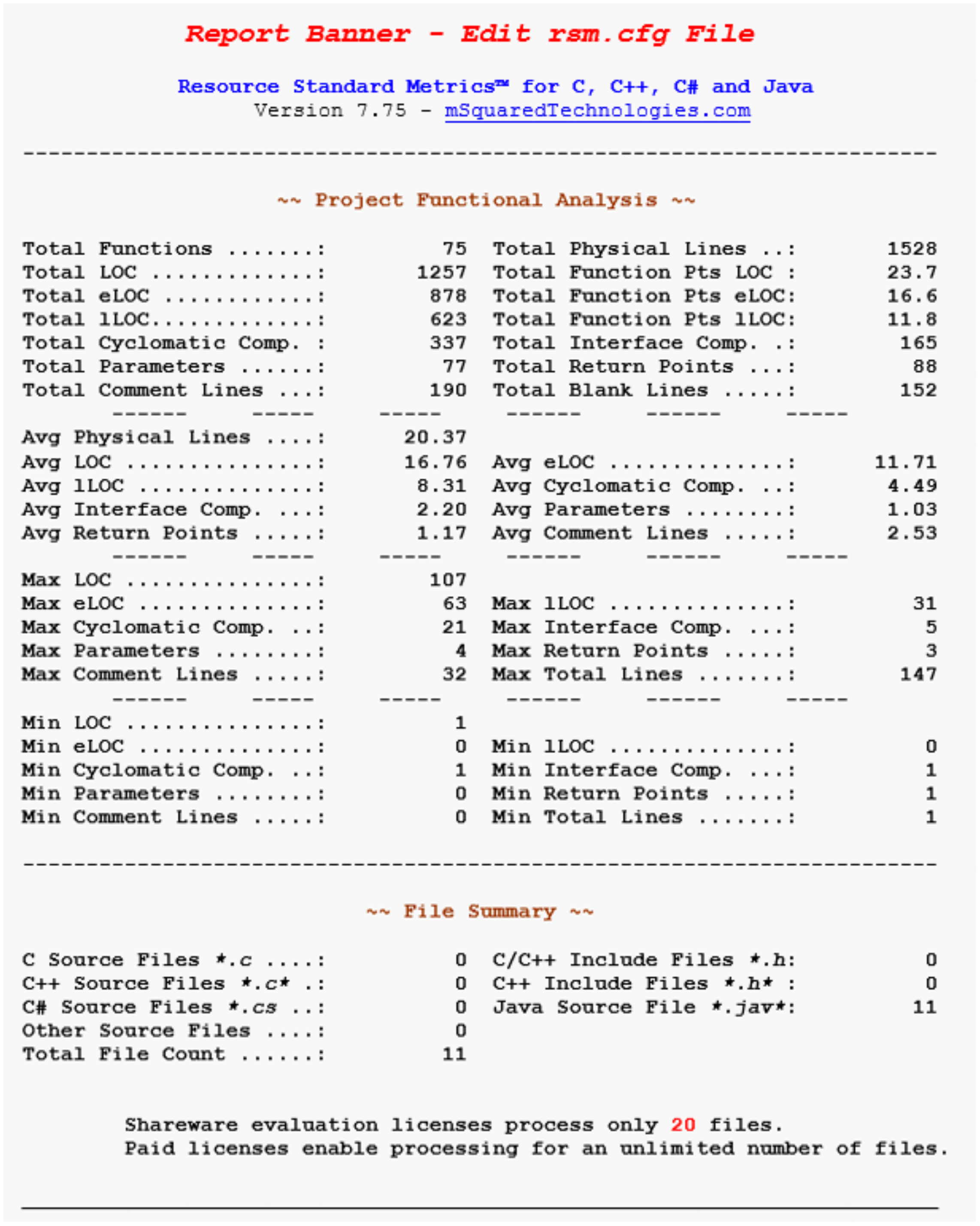
```
                Report Banner - Edit rsm.cfg File

           Resource Standard Metrics™ for C, C++, C# and Java
                Version 7.75 - mSquaredTechnologies.com

  ------------------------------------------------------------------------

                  ~~ Project Functional Analysis ~~

  Total Functions .......:         51  Total Physical Lines ..:        552
  Total LOC .............:        483  Total Function Pts LOC :        9.1
  Total eLOC ............:        260  Total Function Pts eLOC:        4.9
  Total lLOC.............:        188  Total Function Pts lLOC:        3.5
  Total Cyclomatic Comp. :         69  Total Interface Comp. .:         93
  Total Parameters ......:         37  Total Return Points ...:         56
  Total Comment Lines ...:         26  Total Blank Lines .....:         45
         ------     -----     -----     ------     ------     -----
  Avg Physical Lines ....:      10.82
  Avg LOC ...............:       9.47  Avg eLOC ..............:       5.10
  Avg lLOC ..............:       3.69  Avg Cyclomatic Comp. ..:       1.35
  Avg Interface Comp. ...:       1.82  Avg Parameters ........:       0.73
  Avg Return Points .....:       1.10  Avg Comment Lines .....:       0.51
         ------     -----     -----     ------     ------     -----
  Max LOC ...............:         38
  Max eLOC ..............:         22  Max lLOC ..............:         16
  Max Cyclomatic Comp. ..:          4  Max Interface Comp. ...:          4
  Max Parameters ........:          3  Max Return Points .....:          3
  Max Comment Lines .....:          6  Max Total Lines .......:         50
         ------     -----     -----     ------     ------     -----
  Min LOC ...............:          1
  Min eLOC ..............:          0  Min lLOC ..............:          0
  Min Cyclomatic Comp. ..:          1  Min Interface Comp. ...:          1
  Min Parameters ........:          0  Min Return Points .....:          1
  Min Comment Lines .....:          0  Min Total Lines .......:          1

  ------------------------------------------------------------------------

                         ~~ File Summary ~~

  C Source Files *.c ....:          0  C/C++ Include Files *.h:          0
  C++ Source Files *.c* .:          0  C++ Include Files *.h* :          0
  C# Source Files *.cs ..:          0  Java Source File *.jav*:         13
  Other Source Files ....:          0
  Total File Count ......:         13


          Shareware evaluation licenses process only 20 files.
          Paid licenses enable processing for an unlimited number of files.
```

Figure C.3. RSM report for project C1 (part 3 of 3) – from SerialSensor.java to TCPServer.java.



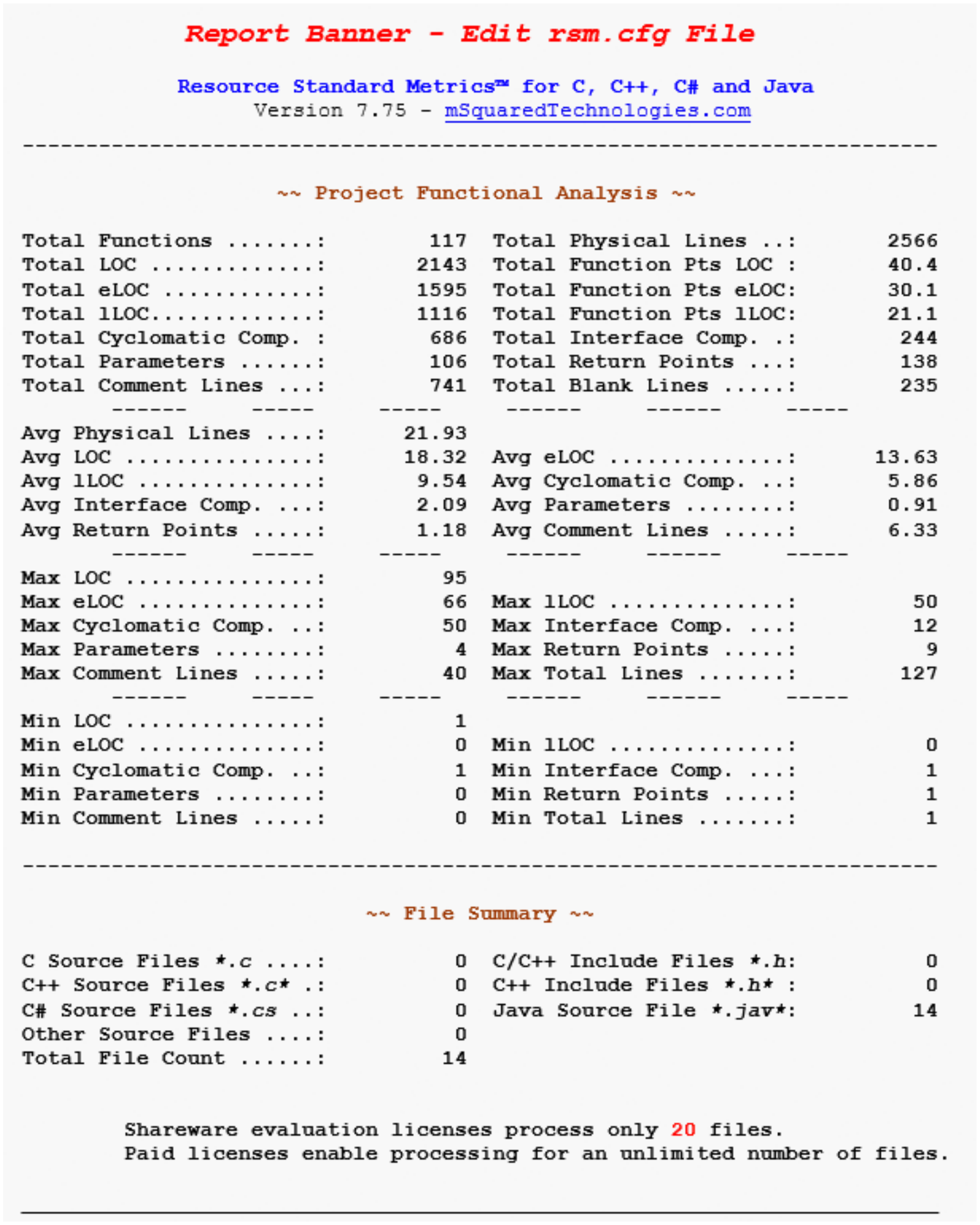
```
                Report Banner - Edit rsm.cfg File

          Resource Standard Metrics™ for C, C++, C# and Java
                Version 7.75 - mSquaredTechnologies.com
-----------------------------------------------------------------------

                   ~~ Project Functional Analysis ~~

Total Functions .......:         5  Total Physical Lines ..:       223
Total LOC .............:       181  Total Function Pts LOC :       3.4
Total eLOC ............:       167  Total Function Pts eLOC:       3.2
Total lLOC.............:       140  Total Function Pts lLOC:       2.6
Total Cyclomatic Comp. :        44  Total Interface Comp. .:         6
Total Parameters ......:         1  Total Return Points ...:         5
Total Comment Lines ...:        11  Total Blank Lines .....:        31
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:     44.60
Avg LOC ...............:     36.20  Avg eLOC ..............:     33.40
Avg lLOC ..............:     28.00  Avg Cyclomatic Comp. ..:      8.80
Avg Interface Comp. ...:      1.20  Avg Parameters ........:      0.20
Avg Return Points .....:      1.00  Avg Comment Lines .....:      2.20
       ------     -----     -----     ------     ------     -----
Max LOC ...............:        51
Max eLOC ..............:        48  Max lLOC ..............:        40
Max Cyclomatic Comp. ..:        17  Max Interface Comp. ...:         2
Max Parameters ........:         1  Max Return Points .....:         1
Max Comment Lines .....:         5  Max Total Lines .......:        62
       ------     -----     -----     ------     ------     -----
Min LOC ...............:         3
Min eLOC ..............:         2  Min lLOC ..............:         1
Min Cyclomatic Comp. ..:         1  Min Interface Comp. ...:         1
Min Parameters ........:         0  Min Return Points .....:         1
Min Comment Lines .....:         0  Min Total Lines .......:         4

-----------------------------------------------------------------------

                          ~~ File Summary ~~

C Source Files *.c ....:         0  C/C++ Include Files *.h:         0
C++ Source Files *.c* .:         0  C++ Include Files *.h* :         0
C# Source Files *.cs ..:         0  Java Source File *.jav*:         1
Other Source Files ....:         0
Total File Count ......:         1


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.

```

Figure C.4. RSM report for project R1.



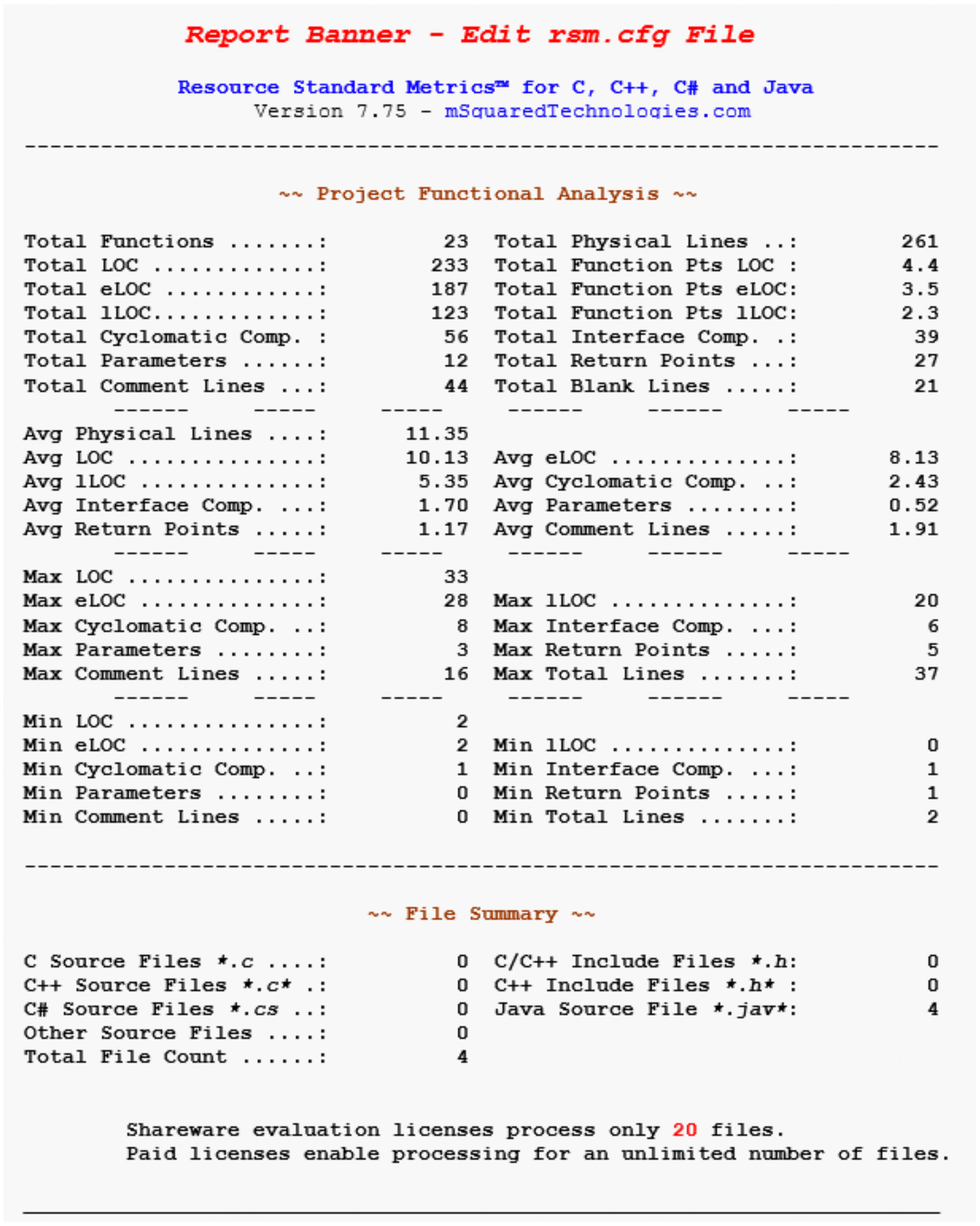
```
                  Report Banner - Edit rsm.cfg File

             Resource Standard Metrics™ for C, C++, C# and Java
                   Version 7.75 - mSquaredTechnologies.com

------------------------------------------------------------------------

                   ~~ Project Functional Analysis ~~

Total Functions .......:         75 Total Physical Lines ..:       1528
Total LOC .............:       1257 Total Function Pts LOC :       23.7
Total eLOC ............:        878 Total Function Pts eLOC:       16.6
Total lLOC.............:        623 Total Function Pts lLOC:       11.8
Total Cyclomatic Comp. :        337 Total Interface Comp. .:        165
Total Parameters ......:         77 Total Return Points ...:         88
Total Comment Lines ...:        190 Total Blank Lines .....:        152
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:      20.37
Avg LOC ...............:      16.76 Avg eLOC ..............:      11.71
Avg lLOC ..............:       8.31 Avg Cyclomatic Comp. ..:       4.49
Avg Interface Comp. ...:       2.20 Avg Parameters ........:       1.03
Avg Return Points .....:       1.17 Avg Comment Lines .....:       2.53
       ------     -----     -----     ------     ------     -----
Max LOC ...............:        107
Max eLOC ..............:         63 Max lLOC ..............:         31
Max Cyclomatic Comp. ..:         21 Max Interface Comp. ...:          5
Max Parameters ........:          4 Max Return Points .....:          3
Max Comment Lines .....:         32 Max Total Lines .......:        147
       ------     -----     -----     ------     ------     -----
Min LOC ...............:          1
Min eLOC ..............:          0 Min lLOC ..............:          0
Min Cyclomatic Comp. ..:          1 Min Interface Comp. ...:          1
Min Parameters ........:          0 Min Return Points .....:          1
Min Comment Lines .....:          0 Min Total Lines .......:          1

------------------------------------------------------------------------

                          ~~ File Summary ~~

C Source Files *.c ....:          0 C/C++ Include Files *.h:          0
C++ Source Files *.c* .:          0 C++ Include Files *.h* :          0
C# Source Files *.cs ..:          0 Java Source File *.jav*:         11
Other Source Files ....:          0
Total File Count ......:         11


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.
```

Figure C.5. RSM report for a freeware implementation of the Pac-Man video game used to create project C3.



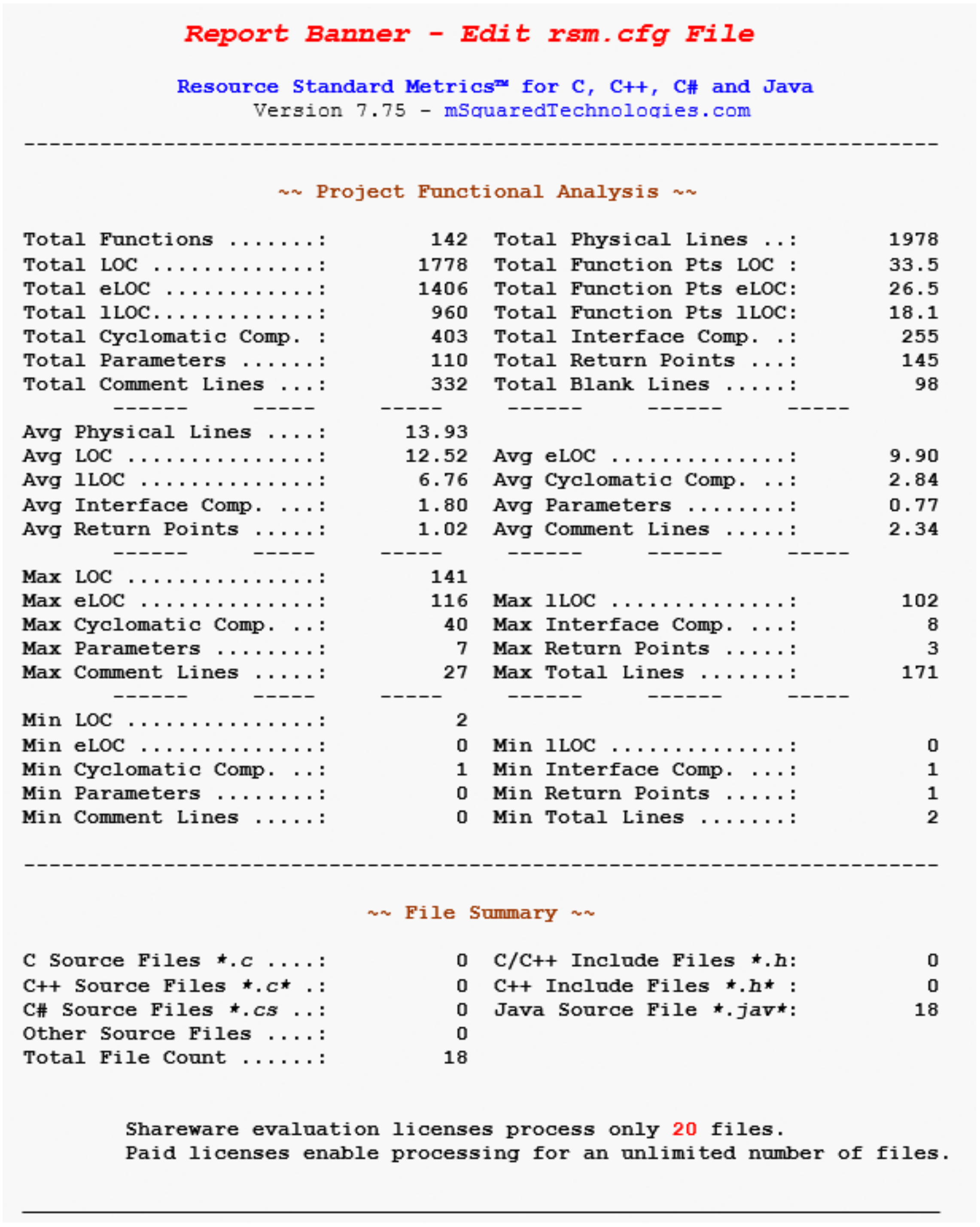
```
                Report Banner - Edit rsm.cfg File

         Resource Standard Metrics™ for C, C++, C# and Java
               Version 7.75 - mSquaredTechnologies.com

------------------------------------------------------------------------

                 ~~ Project Functional Analysis ~~

Total Functions .......:       117 Total Physical Lines ..:      2566
Total LOC .............:      2143 Total Function Pts LOC :      40.4
Total eLOC ............:      1595 Total Function Pts eLOC:      30.1
Total lLOC.............:      1116 Total Function Pts lLOC:      21.1
Total Cyclomatic Comp. :       686 Total Interface Comp. .:       244
Total Parameters ......:       106 Total Return Points ...:       138
Total Comment Lines ...:       741 Total Blank Lines .....:       235
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:     21.93
Avg LOC ...............:     18.32 Avg eLOC ..............:     13.63
Avg lLOC ..............:      9.54 Avg Cyclomatic Comp. ..:      5.86
Avg Interface Comp. ...:      2.09 Avg Parameters ........:      0.91
Avg Return Points .....:      1.18 Avg Comment Lines .....:      6.33
       ------     -----     -----     ------     ------     -----
Max LOC ...............:        95
Max eLOC ..............:        66 Max lLOC ..............:        50
Max Cyclomatic Comp. ..:        50 Max Interface Comp. ...:        12
Max Parameters ........:         4 Max Return Points .....:         9
Max Comment Lines .....:        40 Max Total Lines .......:       127
       ------     -----     -----     ------     ------     -----
Min LOC ...............:         1
Min eLOC ..............:         0 Min lLOC ..............:         0
Min Cyclomatic Comp. ..:         1 Min Interface Comp. ...:         1
Min Parameters ........:         0 Min Return Points .....:         1
Min Comment Lines .....:         0 Min Total Lines .......:         1

------------------------------------------------------------------------

                         ~~ File Summary ~~

C Source Files *.c ....:         0 C/C++ Include Files *.h:         0
C++ Source Files *.c* .:         0 C++ Include Files *.h* :         0
C# Source Files *.cs ..:         0 Java Source File *.jav*:        14
Other Source Files ....:         0
Total File Count ......:        14


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.
```

Figure C.6. RSM report for project C3.



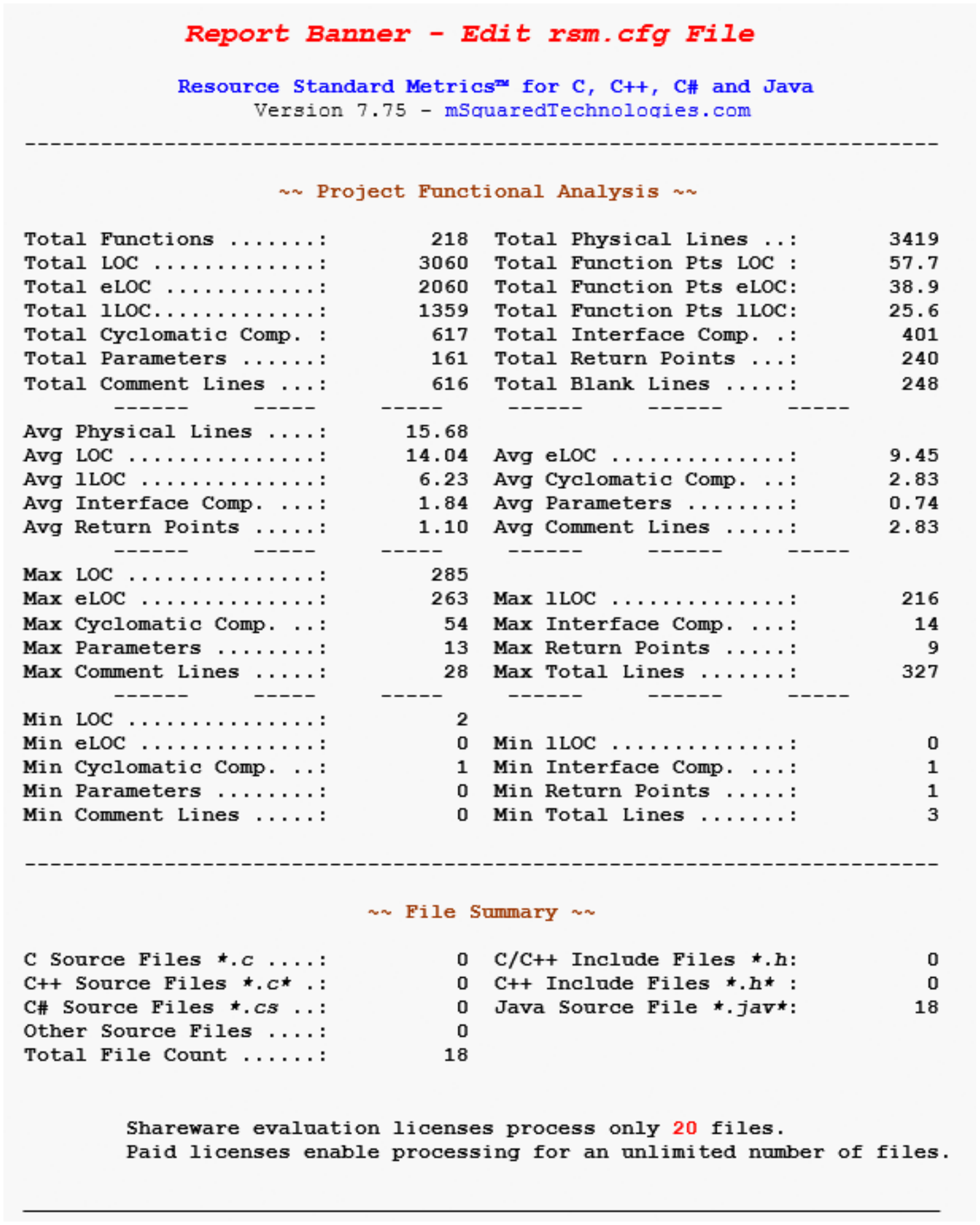
```
                Report Banner - Edit rsm.cfg File

              Resource Standard Metrics™ for C, C++, C# and Java
                    Version 7.75 - mSquaredTechnologies.com

------------------------------------------------------------------------

                    ~~ Project Functional Analysis ~~

Total Functions .......:        23  Total Physical Lines ..:       261
Total LOC .............:       233  Total Function Pts LOC :       4.4
Total eLOC ............:       187  Total Function Pts eLOC:       3.5
Total lLOC.............:       123  Total Function Pts lLOC:       2.3
Total Cyclomatic Comp. :        56  Total Interface Comp. .:        39
Total Parameters ......:        12  Total Return Points ...:        27
Total Comment Lines ...:        44  Total Blank Lines .....:        21
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:     11.35
Avg LOC ...............:     10.13  Avg eLOC ..............:      8.13
Avg lLOC ..............:      5.35  Avg Cyclomatic Comp. ..:      2.43
Avg Interface Comp. ...:      1.70  Avg Parameters ........:      0.52
Avg Return Points .....:      1.17  Avg Comment Lines .....:      1.91
       ------     -----     -----     ------     ------     -----
Max LOC ...............:        33
Max eLOC ..............:        28  Max lLOC ..............:        20
Max Cyclomatic Comp. ..:         8  Max Interface Comp. ...:         6
Max Parameters ........:         3  Max Return Points .....:         5
Max Comment Lines .....:        16  Max Total Lines .......:        37
       ------     -----     -----     ------     ------     -----
Min LOC ...............:         2
Min eLOC ..............:         2  Min lLOC ..............:         0
Min Cyclomatic Comp. ..:         1  Min Interface Comp. ...:         1
Min Parameters ........:         0  Min Return Points .....:         1
Min Comment Lines .....:         0  Min Total Lines .......:         2

------------------------------------------------------------------------

                         ~~ File Summary ~~

C Source Files *.c ....:         0  C/C++ Include Files *.h:         0
C++ Source Files *.c* .:         0  C++ Include Files *.h* :         0
C# Source Files *.cs ..:         0  Java Source File *.jav*:         4
Other Source Files ....:         0
Total File Count ......:         4


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.
```

Figure C.7. RSM report for project C4.



```
                Report Banner - Edit rsm.cfg File

          Resource Standard Metrics™ for C, C++, C# and Java
                 Version 7.75 - mSquaredTechnologies.com

------------------------------------------------------------------------

                   ~~ Project Functional Analysis ~~

Total Functions .......:        142 Total Physical Lines ..:       1978
Total LOC .............:       1778 Total Function Pts LOC :       33.5
Total eLOC ............:       1406 Total Function Pts eLOC:       26.5
Total lLOC.............:        960 Total Function Pts lLOC:       18.1
Total Cyclomatic Comp. :        403 Total Interface Comp. .:        255
Total Parameters ......:        110 Total Return Points ...:        145
Total Comment Lines ...:        332 Total Blank Lines .....:         98
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:      13.93
Avg LOC ...............:      12.52 Avg eLOC ..............:       9.90
Avg lLOC ..............:       6.76 Avg Cyclomatic Comp. ..:       2.84
Avg Interface Comp. ...:       1.80 Avg Parameters ........:       0.77
Avg Return Points .....:       1.02 Avg Comment Lines .....:       2.34
       ------     -----     -----     ------     ------     -----
Max LOC ...............:        141
Max eLOC ..............:        116 Max lLOC ..............:        102
Max Cyclomatic Comp. ..:         40 Max Interface Comp. ...:          8
Max Parameters ........:          7 Max Return Points .....:          3
Max Comment Lines .....:         27 Max Total Lines .......:        171
       ------     -----     -----     ------     ------     -----
Min LOC ...............:          2
Min eLOC ..............:          0 Min lLOC ..............:          0
Min Cyclomatic Comp. ..:          1 Min Interface Comp. ...:          1
Min Parameters ........:          0 Min Return Points .....:          1
Min Comment Lines .....:          0 Min Total Lines .......:          2

------------------------------------------------------------------------

                          ~~ File Summary ~~

C Source Files *.c ....:          0 C/C++ Include Files *.h:          0
C++ Source Files *.c* .:          0 C++ Include Files *.h* :          0
C# Source Files *.cs ..:          0 Java Source File *.jav*:         18
Other Source Files ....:          0
Total File Count ......:         18


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.

------------------------------------------------------------------------
```

Figure C.8. RSM report for package AC of project R2 (part 1 of 2).



```
                 Report Banner - Edit rsm.cfg File

           Resource Standard Metrics™ for C, C++, C# and Java
                Version 7.75 - mSquaredTechnologies.com

------------------------------------------------------------------------

                    ~~ Project Functional Analysis ~~

Total Functions .......:      218 Total Physical Lines ..:      3419
Total LOC .............:     3060 Total Function Pts LOC :      57.7
Total eLOC ............:     2060 Total Function Pts eLOC:      38.9
Total lLOC.............:     1359 Total Function Pts lLOC:      25.6
Total Cyclomatic Comp. :      617 Total Interface Comp. .:       401
Total Parameters ......:      161 Total Return Points ...:       240
Total Comment Lines ...:      616 Total Blank Lines .....:       248
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:    15.68
Avg LOC ...............:    14.04 Avg eLOC ..............:      9.45
Avg lLOC ..............:     6.23 Avg Cyclomatic Comp. ..:      2.83
Avg Interface Comp. ...:     1.84 Avg Parameters ........:      0.74
Avg Return Points .....:     1.10 Avg Comment Lines .....:      2.83
       ------     -----     -----     ------     ------     -----
Max LOC ...............:      285
Max eLOC ..............:      263 Max lLOC ..............:       216
Max Cyclomatic Comp. ..:       54 Max Interface Comp. ...:        14
Max Parameters ........:       13 Max Return Points .....:         9
Max Comment Lines .....:       28 Max Total Lines .......:       327
       ------     -----     -----     ------     ------     -----
Min LOC ...............:        2
Min eLOC ..............:        0 Min lLOC ..............:         0
Min Cyclomatic Comp. ..:        1 Min Interface Comp. ...:         1
Min Parameters ........:        0 Min Return Points .....:         1
Min Comment Lines .....:        0 Min Total Lines .......:         3

------------------------------------------------------------------------

                           ~~ File Summary ~~

C Source Files *.c ....:        0 C/C++ Include Files *.h:         0
C++ Source Files *.c* .:        0 C++ Include Files *.h* :         0
C# Source Files *.cs ..:        0 Java Source File *.jav*:        18
Other Source Files ....:        0
Total File Count ......:       18


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.

------------------------------------------------------------------------
```

Figure C.9. RSM report for package AC of project R2 (part 2 of 2).

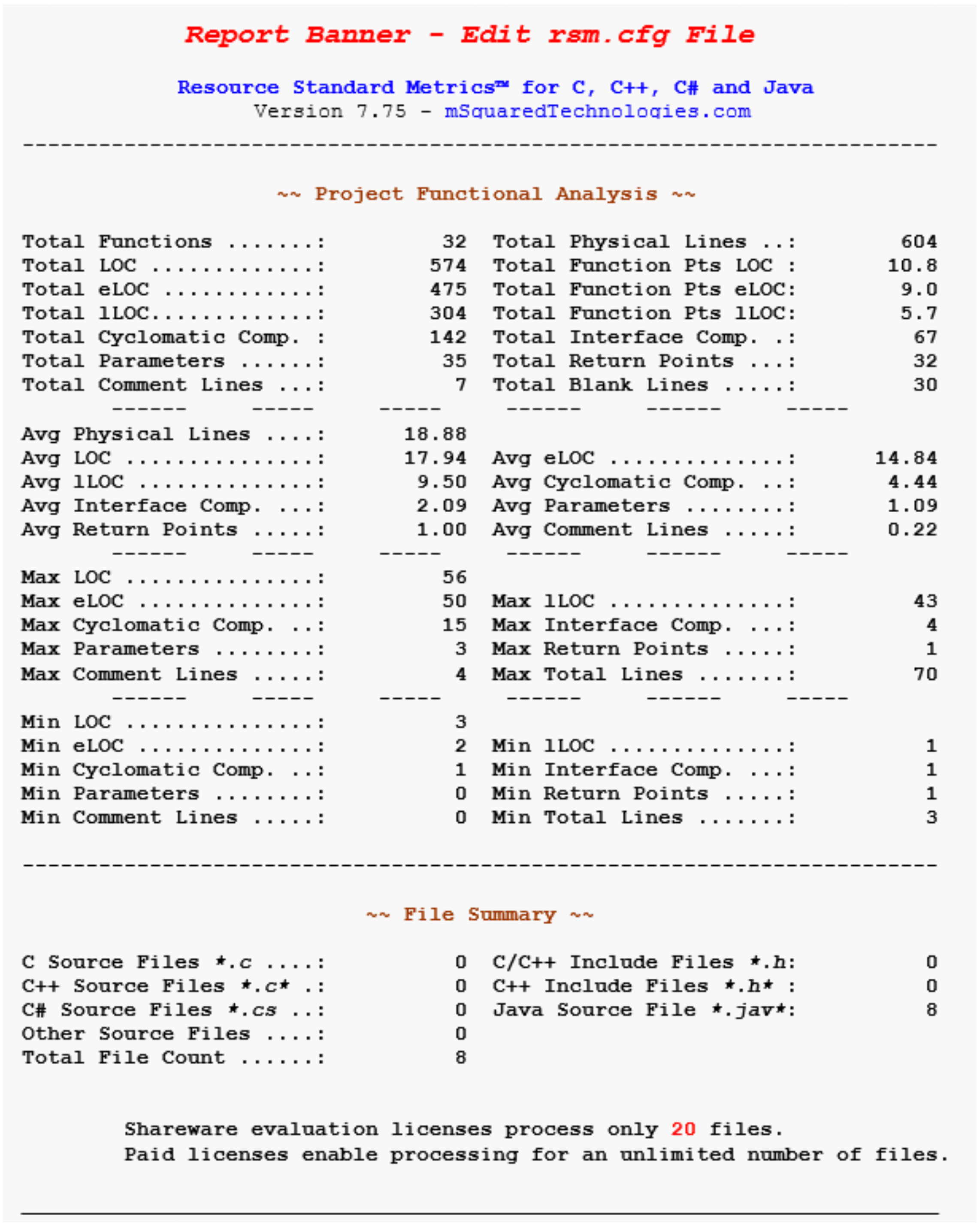
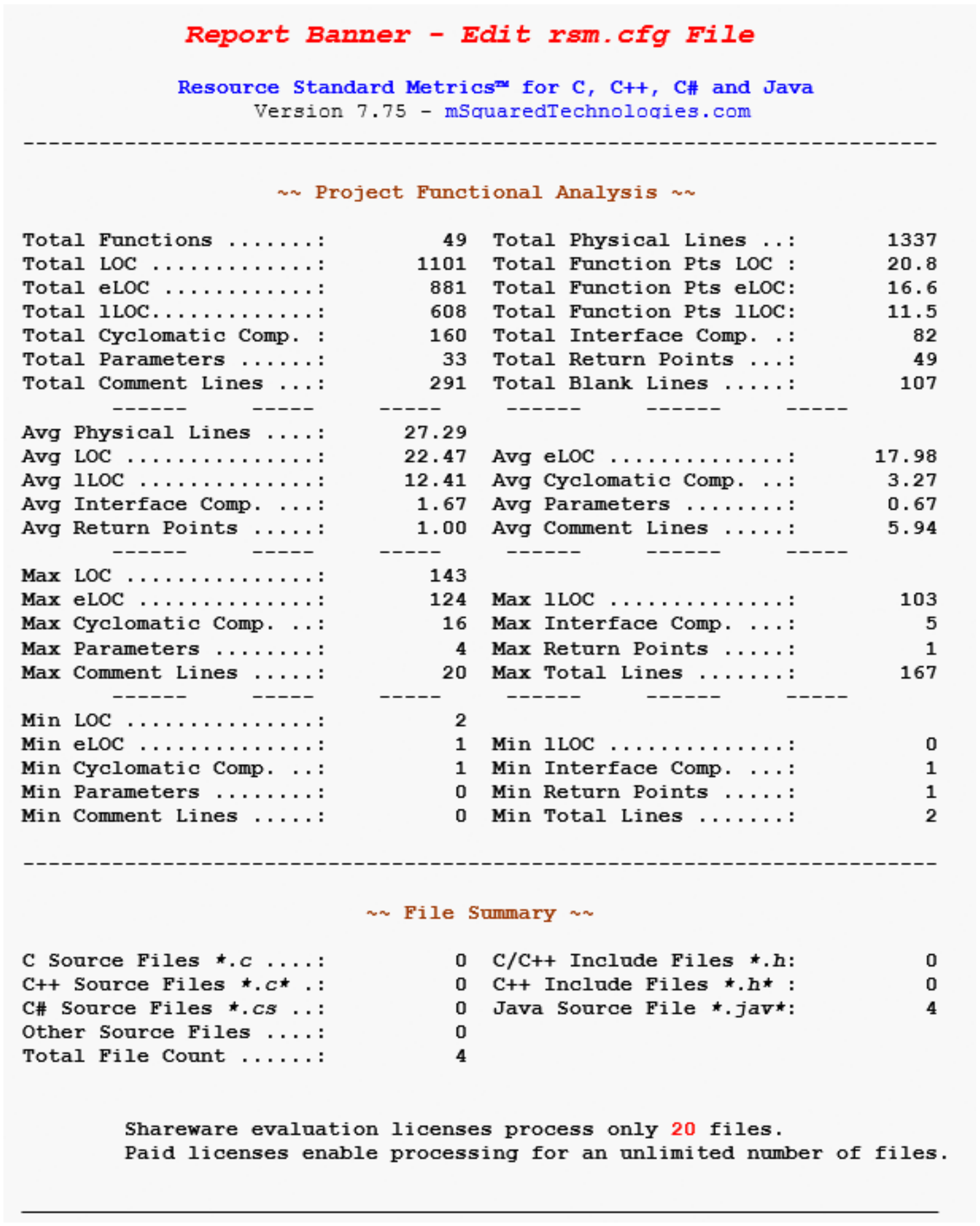

```
               Report Banner - Edit rsm.cfg File

           Resource Standard Metrics™ for C, C++, C# and Java
                 Version 7.75 - mSquaredTechnologies.com

----------------------------------------------------------------------

                  ~~ Project Functional Analysis ~~

Total Functions .......:         32  Total Physical Lines ..:       604
Total LOC .............:        574  Total Function Pts LOC :      10.8
Total eLOC ............:        475  Total Function Pts eLOC:       9.0
Total lLOC.............:        304  Total Function Pts lLOC:       5.7
Total Cyclomatic Comp. :        142  Total Interface Comp. .:        67
Total Parameters ......:         35  Total Return Points ...:        32
Total Comment Lines ...:          7  Total Blank Lines .....:        30
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:      18.88
Avg LOC ...............:      17.94  Avg eLOC ..............:     14.84
Avg lLOC ..............:       9.50  Avg Cyclomatic Comp. ..:      4.44
Avg Interface Comp. ...:       2.09  Avg Parameters ........:      1.09
Avg Return Points .....:       1.00  Avg Comment Lines .....:      0.22
       ------     -----     -----     ------     ------     -----
Max LOC ...............:         56
Max eLOC ..............:         50  Max lLOC ..............:        43
Max Cyclomatic Comp. ..:         15  Max Interface Comp. ...:         4
Max Parameters ........:          3  Max Return Points .....:         1
Max Comment Lines .....:          4  Max Total Lines .......:        70
       ------     -----     -----     ------     ------     -----
Min LOC ...............:          3
Min eLOC ..............:          2  Min lLOC ..............:         1
Min Cyclomatic Comp. ..:          1  Min Interface Comp. ...:         1
Min Parameters ........:          0  Min Return Points .....:         1
Min Comment Lines .....:          0  Min Total Lines .......:         3

----------------------------------------------------------------------

                         ~~ File Summary ~~

C Source Files *.c ....:          0  C/C++ Include Files *.h:         0
C++ Source Files *.c* .:          0  C++ Include Files *.h* :         0
C# Source Files *.cs ..:          0  Java Source File *.jav*:         8
Other Source Files ....:          0
Total File Count ......:          8


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.

______________________________________________________________________
```

Figure C.10. RSM report for the legacy code used in the AC package of project R2.



```
                  Report Banner - Edit rsm.cfg File

                Resource Standard Metrics™ for C, C++, C# and Java
                     Version 7.75 - mSquaredTechnologies.com

------------------------------------------------------------------------

                    ~~ Project Functional Analysis ~~

Total Functions .......:         49  Total Physical Lines ..:       1337
Total LOC .............:       1101  Total Function Pts LOC :       20.8
Total eLOC ............:        881  Total Function Pts eLOC:       16.6
Total lLOC.............:        608  Total Function Pts lLOC:       11.5
Total Cyclomatic Comp. :        160  Total Interface Comp. .:         82
Total Parameters ......:         33  Total Return Points ...:         49
Total Comment Lines ...:        291  Total Blank Lines .....:        107
       ------     -----    -----    ------     ------     -----
Avg Physical Lines ....:      27.29
Avg LOC ...............:      22.47  Avg eLOC ..............:      17.98
Avg lLOC ..............:      12.41  Avg Cyclomatic Comp. ..:       3.27
Avg Interface Comp. ...:       1.67  Avg Parameters ........:       0.67
Avg Return Points .....:       1.00  Avg Comment Lines .....:       5.94
       ------     -----    -----    ------     ------     -----
Max LOC ...............:        143
Max eLOC ..............:        124  Max lLOC ..............:        103
Max Cyclomatic Comp. ..:         16  Max Interface Comp. ...:          5
Max Parameters ........:          4  Max Return Points .....:          1
Max Comment Lines .....:         20  Max Total Lines .......:        167
       ------     -----    -----    ------     ------     -----
Min LOC ...............:          2
Min eLOC ..............:          1  Min lLOC ..............:          0
Min Cyclomatic Comp. ..:          1  Min Interface Comp. ...:          1
Min Parameters ........:          0  Min Return Points .....:          1
Min Comment Lines .....:          0  Min Total Lines .......:          2

------------------------------------------------------------------------

                           ~~ File Summary ~~

C Source Files *.c ....:          0  C/C++ Include Files *.h:          0
C++ Source Files *.c* .:          0  C++ Include Files *.h* :          0
C# Source Files *.cs ..:          0  Java Source File *.jav*:          4
Other Source Files ....:          0
Total File Count ......:          4


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.

```

Figure C.11. RSM report for the legacy code used in the ELE package (affective companion) of project R3.