Unified GUI Adaptation in Dynamic Software Product Lines

Dean Matthew Kramer

A thesis submitted in partial
fulfilment of the requirements of the
University of West London
for the degree of Doctor of Philosophy

September 2014

To the memory of my mother,
Lynn Kramer, 1956-2015
Whose eternal smile and love will always be remembered

Acknowledgements

I have to confess, when I embarked on this PhD, I was only interested in what I felt was the goal, and how best to get it. One of my supervisors always said "*It's not the end that is important, it's the journey*". At the time, I could not really understand how the journey could be more important than the end. However, having completed this PhD, I now understand that message. On this journey, as all great journeys, important people always help shape it. For this, I feel deep gratitude is required.

Firstly, I wish to thank Dr. Samia Oussena, Prof. Peter Komisarczuk, and Prof. Tony Clark. A special thanks to Samia, for all the opportunities, guidance, and continual useful critiques of my research. Many thanks to Peter, your advice, suggestions and positivity have always been important. A big thanks to Tony, for being my external supervisor and your invaluable feedback and support.

I would like to thank all my fellow PhD students, particularly Dr. Anna Kocúrová and Malte Reßin. You both were so important during the good times, and the bad. Thank you for making this journey more fun, and in many cases, bearable. Thank you also to Antonio Khierkhazadeh, Jiva Bagale, and Christian Sauer for the great research discussions and advice.

I wish to express my thanks to all members of the School of Computing and Technology at the University of West London. I especially wish to give my deep thanks and respect to the late Prof. Andy Smith for his support, and encouragement at the beginning of my PhD. I also wish to give thanks to Prof. Thomas Roth-Berghofer for all the valuable input, and idea inspiring research discussions. Thank you also to Dr. John Moore for always being a positive role model during my time as a research assistant and PhD student.

I would also like to thank all my non-academic friends back on Canvey Island. Thank you for always being behind me through the years!

Finally, I would like to thank my family. First my parents, for your sacrifices and selfless love. I know I have not always been an easy son. My brother, Grant, for his constant words of encouragement, and for always being a great brother. To my partner, Katarína, thank you for your love, support, and always being there for me. Lastly, I would like to give a special thanks to my Mum for everything she ever did for me. I really could not have asked for a better Mother. There is an old proverb that says "*God couldn't be everywhere, so He created mothers*", which for me, could not be more true.

Abstract

In the modern world of mobile computing and ubiquitous technology, society is able to interact with technology in new and fascinating ways. To help provide an improved user experience, mobile software should be able to adapt itself to suit the user. By monitoring context information based on the environment and user, the application can better meet the dynamic requirements of the user. Similarly, it is noticeable that programs can require different static changes to suit static requirements. This program commonality and variability can benefit from the use of Software Product Line Engineering, reusing artefacts over a set of similar programs, called a Software Product Line (SPL). Historically, SPLs are limited to handling static compile time adaptations. Dynamic Software Product Lines (DSPL) however, allow for the program configuration to change at runtime, allow for compile time and runtime adaptation to be developed in a single unified approach. While currently DSPLs provide methods for dealing with program logic adaptations, variability in the Graphical User Interface (GUI) has largely been neglected. Due to this, depending on the intended time to apply GUI adaptation, different approaches are required. The main goal of this work is to extend a unified representation of variability to the GUI, whereby GUI adaptation can be applied at compile time and at runtime.

In this thesis, an approach to handling GUI adaptation within DSPLs, providing a unified representation of GUI variability is presented. The approach is based on Feature-Oriented Programming (FOP), enabling developers to implement GUI adaptation along with program logic in feature modules. This approach is applied to Document-Oriented GUIs, also known as GUI description languages. In addition to GUI unification, we present an approach to unifying context and feature modelling, and handling context dynamically at runtime, as features of the DSPL. This unification can allow for more dynamic and self-aware context acquisition. To validate our approach, we implemented tool support and middleware prototypes. These different artefacts are then tested using a combination of scenarios and scalability tests. This combination first helps demonstrate the versatility and its relevance of the different approach aspects. It further brings insight into how the approach scales with DSPL size.

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

Contents

1.1 Motivation

The high proliferation of mobile computing devices including smart phones and tablets displays a shift in how people interact with computers, realising Weiser (1991) vision of ubiquitous computing. This shift is allowing people to stay connected and active without needing to be in a fixed location. These mobile computing devices are heterogeneous in different ways, including their physical hardware specifications, and the software that executes on them.

Along with this high mobile device market penetration brings a boom in the number of mobile applications being developed and sold to users. Applications developed for modern smart devices are often highly responsive, and include a rich user interface using different graphical user interface (GUI) design techniques, and physical gesture support. These together help provide the user with a quicker, and more intuitive tool for working, and leisure. Because these applications are consumed by people from different countries, cultures, and possible handicaps, some form of personalisation is required. This personalisation includes internationalisation, and localisation which can require language changes, and other non trivial visual changes (Russo and Boor, 1993). Without useful and appealing GUIs, applications are likely to be purchased and used less by users.

A benefit of mobile computing devices, and their applications includes the ability to continue to be active in dynamic conditions, including one's physical location. While different hardware elements of the device can and do differ, it is very common for these devices to include different sensors to monitor the changing conditions of the user and device. These sensors include accelerometers, GPS, light sensors etc. Using these sensors and different available internet resources can allow the mobile applications to gain a far better understanding of the current situation of the user, and potential intentions and requirements. Intelligent applications that take into account situation of the user, defined as context, are known as *context-aware applications* (Daniele et al., 2009). Examples of these applications include mobile tour guides (Abowd et al., 1997), reminder systems (Dey and Abowd, 2000), and interruption management systems (Avrahami and Hudson, 2006). Using context-awareness, application GUIs can adapt to suit the new context the user or device is in. These adaptations can include background logic in the application, which are not always explicitly noticeable to the user. Other adaptations can include the GUI using adaptive GUIs.

Adaptive GUIs though have seen much research activity over the years, whereby the GUI can be automatically adapted to suit changing runtime requirements, or to suit a particular user context. This can assist the user, and improve application usability. A problem however with adaptive GUI approaches is that they do not follow SPL approaches, and therefore adaptation can only be made at runtime. This consequently means that depending on the time of adaptation, GUI variability will need to be designed and implemented differently. Other issues includes the ability for tailoring these adaptations for different users. Tailoring applications without systematic reuse leads the developer towards developing different program assets more than once.

Developing software products is often expensive and time intensive. Due to highly variable nature of mobile computing devices, and user requirements, developing and maintaining products to accommodate this variability is a difficult task. By using *Software Product Line Engineering* techniques, this variability can be handled in a far more efficient and reusable way. Recent work has also highlighted how variability within the GUI including its design and behaviour can be high (Pleuss et al., 2012b). These can be to deal with static GUI requirements, including branding for enterprises, and can be to handle different cultural requirements for example right-to-left and left-to-right designs for different cultures and languages. Using SPLs to handle GUI variability enables the developers achieve higher asset reuse.

As described earlier, while developers need to develop applications that can be statically applied to different user groups and requirements, they also often need to deal with changing dynamic conditions of the user and device. In essence, these two types of variability carry out the same objective: to adapt the application in a particular way to suit a new user requirement. Therefore, it is ideal for the developer if static and dynamic adaptations need only one implementation, which can be statically or dynamically applied. To fit this ideal, Dynamic Software Product Lines (DSPL) have been proposed (Hallsteinsen et al., 2008). DSPLs can be driven both by manual product reconfiguration, or also automatically using context-awareness (Parra et al., 2009).

1.1.1 Challenges of Dynamic Software Product Lines

The ability of a SPL to reconfigure at runtime is what defines it as a DSPL. As such, program adaptations and logic can be applied at different points, depending on dynamic user requirements. Historically, DSPLs have concentrated on source code logic adaptation, whereby different program logic in a given operation is executed based on the current configuration. The Graphical User Interface (GUI) however, and what adaptations that maybe needed, have largely been neglected. For a developer to implement a GUI that can be adapted statically, and dynamically, this can require the implementation of GUI variability using a combination of approaches. Approaches for handling GUI variability in static SPLs have been proposed (Hauptmann, 2010), in addition to a number of dynamic approaches (Criado et al., 2010; Sottet et al., 2008; Hanumansetty, 2004). This presents a problem when certain variability might be required statically and dynamically in different final products, as it requires the implementation of adaptation twice. This can add considerable time and complexity to the product development and maintenance, which a single unified approach can avoid.

Furthermore, in recent years, the use of Document-Oriented GUIs (DOGUI) (Draheim et al., 2006; Kim and Lutteroth, 2009) has became an adopted and encouraged approach for GUI implementation in mainstream mobile platforms. These documents are predominantly in markup languages. This is to help encourage the separation of GUI design from rest of the program logic using design patterns including the Model-View-Controller. While DOGUIs have became a popular approach to designing GUIs, there is yet an approach that accommodates them in either a SPL, or an adaptive GUI. This has the effect where the developer needs to program manually all GUI visual properties. This process is far longer to undertake as these GUIs can often not be designed in a visual tool.

This research aims to investigate how to effectively handle static, and dynamic changes to the UI in a single unified approach using DOGUIs. Next, to help the reader and to illustrate the research contribution, we introduce a scenario application for use in this thesis.

1.1.2 A Scenario Application

Let us consider a mobile content store application as a scenario application, like the Google Play store. Alice is a 10 year old girl in the United Kingdom that takes her tablet to school as an education tool. The application on her tablet is designed to suit children in school by providing a more fun and appealing GUI. While at school, she downloads different books to her tablet for her to read including children's story books, and educational books. When she gets home, she can practice her different acquired skills and knowledge applications to test her, and help her improve. Alice also enjoys watching videos from the content store. However, because her tablet lacks a large amount of storage space, videos can only be streamed to her device when she has a good data connection including wifi, 4G or 3G.

Tomáš is a 34 year old man from Slovakia that uses the content store on his mobile phone to download games, applications and books. He would like to also download and stream music and video, but currently this is not available to Slovakia due to licensing restrictions from content owners. However, when travelling to some countries in the EU including France, and the United Kingdom he can buy and download this content. As he has the adult version of the content store, the GUI of the application is more suited to older audiences using a minimalistic design. As Tomáš has an expensive new phone with a large amount of storage space, when he is in other countries, he downloads this content so he can access it when he returns home.

As introduced with these different use scenarios, we can see that there are static, and dynamic user requirements.

The remainder of this introduction chapter is structured as follows: Section 1.2 we motivate the problems for which this thesis tackles. In Section 1.3 we summarise the main contributions of this dissertation. Next, Section 1.4 describes the research methodology of this work. Finally, Section 1.5 presents the overall thesis structure.

1.2 Problem Statement

Dynamic Software Product Lines have shown to the ability to provide a unified solution to tackle the need for both compile-time and runtime adaptation. There are still existing issues that remain regarding these types of systems including *user interface variability*, and *context design, and management*.

1.2.1 User Interface Variability

The GUI just like the rest of the software system can exhibit variability (Pleuss et al., 2012b). This means different features of the system may crosscut the user interface in terms of its representation and/or functionality. When developing GUIs using documents or GUI description languages, there are multiple artefact types that can contain this variability. Dealing with this variability at design time can be handled using current SPL techniques and tools. However, for runtime adaptation, this has yet to be achieved with SPLs, and is only tackled in using adaptive GUI approaches.

Adaptive GUIs though, are not conventionally developed using SPL techniques and often lack the ability to apply the adaptation statically. As a consequence, the developer is then required to use different approaches for developing GUI variability, be it static, or dynamic. While the adaptation is being applied at different times, they both have the same objective, which is to modify the GUI by adding, removing, or altering particular aspects of the GUI.

Because this adaptation focusses on the same objective, there should be a single unified development of this adaptation, whereby the developer need not use different techniques to achieve adaptation. By enabling unified development of GUI adaptations, the developer can choose after implementation where that adaptation should be applied, statically, or dynamically at runtime. This added flexibility can then provide higher feature cohesion, as adaptation binding times can be applied both to GUI adaptation and logic simultaneously.

1.2.2 Other issues

Context Design and Management

When modelling variability in a SPL, feature modelling is a very common notation. With context modelling, many different notations have been proposed in the past. Historically, DSPL methodologies have predominately used different notations for modelling context and the features of the system. This requires two different notions to be used, and requires a method of bridging context with the system it affects. By modelling both context and the features of the system using a single notation, the developer can then express the context-aware system in a unified model.

Not only should context be modelled in a unified way, it should be handled this way also. Context acquisition that is separated from the system is often static, and is therefore not context-aware. Depending on a given context, it might be applicable for different contexts to be acquired, or not. By considering context as a feature of the system, its acquisition can then be treated like other software features in the system, enabling static, and dynamic binding.

1.2.3 Research Aim and Objectives

This research aims to bring GUI variability realisation to DSPLs, targeting the challenges discussed in the problem statement. By bringing GUI variability to DSPLs, GUI adaptation should be able to be viewed and realised in a single unified way. This adaptation then should be able to be used either at compile time, or at runtime. We also look at how context, which is often used in DSPLs, can be reused over many DSPLs at runtime, allowing for context sharing over applications. With this research aim in mind, our main research objectives can be described as:

1. To investigate and extend current source code refinement approaches to include GUI representation in GUI documents.

2. To investigate and design a runtime mechanism to be capable of runtime adaptation of the GUI using GUI documents.

3. To extend current variability models to include context information, and allowing for dynamic context acquisition.

4. To implement development tool support for variability modelling, and product derivation.

5. To implement a centralised DSPL management system on a mobile platform.

1.2.4 Research Questions

Based on the aim of this research, this thesis attempts to answer the following research questions:

Reseach Question

> ***How can compile time and runtime GUI adaptation be unified within a DSPL?***

By unifying compile-time and runtime GUI adaptation, the developer can then implement the given adaptation once. Following this, the decision on where to apply this adaptation can be chosen, and changed, without the need for reimplementation. Chapter 4 provides a high level view of adaptation, with different types of adaptation. In Chapter 5, a description to how GUI adaptation is defined and implemented, with its runtime behaviour being described in Chapter 6.

Subquestions

- ***How can GUI adaptation be applied dynamically at runtime?***

  After GUI adaptation has been implemented, support is needed to allow for the adaptation to be applied on reconfiguration. In Chapter 6, this question is addressed, showing how GUI adaptation can be applied using variants produced in Chapter 5.

- ***How can context be modelled, and treated by the system as a feature of the DSPL?***

  Context modelling is carried out in feature modelling, described in Chapter 5. This enables the developer to use a single modelling notation for the system. Runtime context handling as a feature is then addressed in Chapter 6, enabling contexts to become context aware.

- ***How can contexts be reused by multiple running DSPL applications?*** Context reuse is described in Chapter 6. Contexts can be used both by multiple DSPLs and other applications on the system. A description of how at runtime new contexts can be deployed to the context management system, and then be used by other applications.

1.3 Contribution

By answering the stated research questions, this thesis makes a number of contributions including:

- **An extended modelling approach to handle system variability and context:** When designing a DSPL, both the system and context that can affect the configuration need to be model. To avoid supplementary models, we use extended feature models, allowing the developer to design their system and context model within a single notation.

- **A Feature-Oriented approach to GUI adaptation:** We propose an approach for implementing GUIs in Dynamic Software Product Lines. GUI variability is implemented within refinements, following Feature-Oriented Programming (FOP). Within each refinement, we propose a hooking method to assist in refinement placement. Compile-time composition is handled using superimposition, in a stepwise fashion. Along side GUI document refinements, we describe an approach that enables other source code GUI adaptations that require execution on reconfiguration, using FOP.

- **A mechanism for runtime GUI adaptation:** After the developer implements GUI adaptation, this can be applied dynamically at runtime. To facilitate visible runtime GUI adaptation, a mechanism is proposed. This mechanism handles automatic adaptation of the GUI, updating only widgets requiring adaptation. It also handles widget state retention to ensure no important state data is lost in the adaptation. This adaptation can be applied at two stages in a GUIs lifecycle, allowing for greater control of when an adaptation is suitable. The architecture, components, and algorithms needed to apply adaptation from GUI documents is defined in this thesis.

- **A centralised DSPL management system:** To assist in managing each of the DSPLs that might be running on a mobile device, a management system is proposed. This management system can manage multiple running DSPLs simultaneously, and allows the reuse of runtime system components over multiple applications. Along with this system, we include a context management system for handling context acquisition and sharing. Using the context model, contexts can become active or inactive dynamically at runtime, just like any another feature.

- **Tool support and developed prototypes:** In the process of validating the contribution of this dissertation, several implementation deliverables have been created. First, we provide tool support as an extension to an open source SPL tool for context modelling, and context rule formulation. This tool support also supports the auto generation of components needed for runtime GUI adaptation, and software composition for the DSPL. Second, a complete software middleware for managing DSPL application configurations was developed for the Android mobile platform. This middleware includes a context management system which allows for context acquisition, and reconfiguration based on context.

The designed technology and methods intends to extend DSPLs to not just program logic, but to GUI representation.

1.3.1 Publications

During the development of this thesis, the author has published in international conferences and workshops, numbered in chronological order:

1. Kramer D., Oussena, S., Komisarczuk, P., Clark, T. (2013) Document-Oriented GUIs in Dynamic Software Product Lines. *In the Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences.*

2. Kramer D., Sauer, C., Roth-Berghofer, T. (2013) Towards Explanation Generation using Feature Models in Software Product Lines. *In the Proceedings of the 9th Workshop on Knowledge Engineering and Software Engineering.*

3. Kramer, D., Oussena, S., Clark, T., Komisarczuk, P. (2013) Graphical User Interfaces in Dynamic Software Product Lines. *In the Proceedings of the 4th International Workshop on Product Line Approaches in Software Engineering.*

4. Sauer, C., Kocurova, A., Kramer, D., Roth-Berghofer, T. (2012) Using canned explanations within a mobile context engine. *In the Proceedings of the 7th Workshop on Explanation-aware Computing.*

5. Kramer, D., Kocurova, A., Oussena, S., Clark, T., Komisarczuk, P. (2011) An extensible, self contained, layered approach to context acquisition. *In the Proceedings of the 3rd International Workshop on Middleware for Pervasive Mobile and Embedded Computing.*

1.4 Research Methodology

Design Research (DR) and Software Engineering Research (SER) function differently to traditional scientific research, by which traditional science studies existing phenomena, while DR and SER study *how* to do things and *how* to create things (Marcos, 2005). This research can be broken down into a number of phases:

- Within phase 1, exploration of the research area is carried out, with an analysis of the current methods for creating DSPLs for mobile systems. This primarily conducted via literature survey, and by experimentation with current methods in the new setting. This phase should end with a final research definition with questions and boundaries being set.
- In phase 2, using a iterative process of implementation and testing, investigation into how GUI elements of a program can be handled within a DSPL are carried, along with how context can be more easily modelled, and used within a DSPL. This research can be viewed as observing a constructivist view, by which that truth and meaning is constructed from our engagement with the world, instead of it being objectively discovered (Feast and Melles, 2010).
- Finally, in phase 3, an evaluation of the outputs from phase 2 is carried out using a combination of scenarios, and scalability tests. These scalability tests will be used to judge how applicable our approach can be for different SPL sizes.

1.5 Thesis Roadmap

In this thesis, there are four main components, *State of the Art, Contribution, Validation* and *Conclusions*. These components are broken down into the following chapters:

1.5.1 Part I: State of the Art

- **Chapter 2: Background.** This chapter introduces Software Product Lines, Context-Awareness, and Graphical User Interface Engineering. In this chapter, we attempt to give a foundation of understanding to some of the concepts and technologies used throughout this piece of work.

- **Chapter 3: Dynamic SPLs and Adaptive GUIs.** This chapter surveys, and discusses the state of the art and related works of this dissertation. We introduce Dynamic Software Products as an approach for developing adaptive software, and their properties. Next we discuss research related to context-awareness and its use in DSPLs, and lastly we consider adaptive GUIs, and how GUIs are handled in SPLs. Finally we set out a concrete set of objectives for this dissertation, based on gaps in knowledge found in the reviewed work.

1.5.2 Part II: Contribution

- **Chapter 4: GUI Variability.** Within this chapter, we define different types of variability that can exist within the system. Following this an abstract specification of the system is proposed, along with requirements for runtime adaptation.

- **Chapter 5: Design Phase.** In this chapter we define the design time stages of the DSPL development, including variability design, implementation, and derivation.

- **Chapter 6: Runtime Phase.** This chapter presents the runtime architecture of the DSPL applications. We present how context is acquired and used for system reconfiguration. Following this, the process of GUI reconfiguration is presented.

1.5.3 Part III: Validation

- **Chapter 7: Implementation.** This chapter presents implementation details of the tool support and middleware software prototype implemented for the Android platform.

- **Chapter 8: Evaluation.** In this chapter, we evaluate the proposed approach using the implementation presented before using scenario SPLs and scalability tests.

1.5.4 Part IV: Conclusion

- **Chapter 9: Conclusions and Future Work.** This chapter presents the final conclusions of the work featured in this thesis. We lastly present motivated work that should be further researched within the field.

Part I

State of the Art

2 Background

Contents

2.1 Introduction

In this chapter we introduce several aspects related to the work within this dissertation. The main goal of this chapter is to introduce a base of knowledge, that can be used throughout this thesis.

The structure of the chapter is as follows: Section 2.2 introduces Software Product Line Engineering. In Section 2.3, we introduce context-aware and adaptive applications. Finally, in Section 2.4, we introduce engineering concepts and methods used for implementing Graphical User Interfaces in applications. Finally, we finish the chapter summarising the different presented approaches.

2.2 Software Product Line Engineering

Software engineering traditionally involves developing singular and individual software systems. In practice though, there are often needs for variation in software to help suit different requirements when developing for different customers. Because of this variability, it has been viewed that it is beneficial to study common properties of programs (Parnas, 1976). Product lines in other industries for example car manufacturing help give greater choice to the customer, where different requirements of the product e.g. colour, number of doors etc can be handled without a car needing to be designed from scratch. Software Product Lines aim at achieving this level of customisation in software systems.

2.2.1 Software Product Lines

Software product lines (SPL) focus on the development of similar software systems using common artefacts. In this thesis, we use Clements and Northrop (2001)'s definition of a Software Product Line as:

> *a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Using SPLs can bring multiple benefits including cost reduction and short time to market, when carried out over multiple products (Pohl et al., 2005). Cost reduction is achieve through the systematic reuse of common artefacts within each product. After each SPL artefact has been produced, they can be reused in a number of other products. When considering each individual application developed traditionally requires every artefact to be developed from scratch, and each artefact has an attached cost to its development, being able to reuse this can gradually lower the total cost of development of a product. Time to market is also reduced because of the reduction of implementation effort required to develop a new product variant because of artefact reuse. For these benefits theough using an SPL, a certain number of products need to be produced, because up front costs can be seen to be higher than individual development. Empirical studies have shown cost break even point, the point where using an SPL costs no more than individual development, is usually around three systems (Weiss and Lai, 1999). This though does depend on many factors of the organisation and products involved.

Success stories of using SPLs instead of traditional software development have been documented (Clements and Northrop, 2001; Pohl et al., 2005). Examples of improvements caused by the use of SPLs include Nokia being able to produce 30 different phones per year, instead of 4; Motorola realising a productivity improvement of 400 %; and HP reporting a factor of seven time to market, and factor of six productivity increase (Bass et al., 2003).

2.2.2 SPL Approaches

There are several approaches for developing a software product line including (Krueger, 2002):

- Proactive SPL development involves the analysis, design, and implementation of the full SPL at once. All foreseeable features requirements are analysed, designed, and implemented. This approach involves heavy initial investment and time, which can prove difficult by small for medium sized companies (Alves et al., 2005).
- Reactive SPL development involves the incremental growth of the product line, in light of new requirements and product demands. Features are designed, and implemented when needed. This approach can offer a faster and more agile approach to an SPL. Also, because of its ability to implement features when needed, this can enable much lower initial costs, as discussed easier.
- Extractive SPL development involves the extractions of commonality and variability within existing software and either develop it into an SPL, or adding to an existing product line. With this approach, existing legacy systems can be decomposed into more reusable assets, which may improve overall system extensibility and maintenance.

While these approaches suit different development circumstances, they can be combined. When considering common software evolution, it may be popular to use a proactive/extractive approach initially, but then move to a reactive approach afterwards.

2.2.3 SPL Processes

Software Product Lines aim to produce mainly similar products using common assets. As such, this means that the development methodology while having some similarities to single product development, still will have differences. While there are some minor discrepancies in different proposed methodologies, there is a common understanding of two engineering processes involved in SPLs, domain and application engineering (Clements and Northrop, 2001; Czarnecki and Eisenecker, 2000).

Domain Engineering

Domain engineering specifically is aimed at the design, realisation, and testing of a re-usable platform. This process is broken up in three distinct parts; *analysis, design,* and *implementation* (Czarnecki and Eisenecker, 2000).

Domain Analysis This process aims to develop and document the domain requirements. This process includes the analysis of domain commonality and variability, that is, what will be reused across multiple applications, and what is likely to differ in those applications. Different modelling techniques for expressing this commonality and variability of an application domain include the *feature-oriented domain analysis* (Kang et al., 1990) and *variability models* (Pohl et al., 2006; Pohl et al., 2005). Using feature-oriented domain analysis, the domain is modelled in terms of system features.

Domain Design. The aim of this process is to develop the domain/reference architecture.

Domain Implementation Within this process, the reusable assets are implemented by the SPL engineer. Many different implementation techniques have been used to implement the SPL variability including feature-oriented programming, aspect-oriented programming, components, and service-oriented architecture.

Application Engineering

As stated earlier, the aim of application engineering is to derive a distinct product from the platform developed in the domain engineering. Just like domain engineering, this process features three distinct parts; *analysis, design* and *implementation*.

Application Analysis This process aims to develop the requirement documents for a given application. Within this process, as much of the requirements from the domain requirements engineering are reused.

Application Design. In this process, the domain/reference architecture designed in the domain engineering phase is specialised. The application architecture is altered according to the application requirements.

Application Implementation. Application implementation is the process of deriving the reusable software artefacts developed in the domain realisation process. A final product is then generated or composed of these software artefacts. Finally, any specific product changes that maybe required are handled here, for example, final customisations of the GUI (Pleuss et al., 2012a).

Commonality and Variability

SPLs are made up of product commonality and variability. Commonality of the product has been described as features that are found in each application from the SPL. Variability on the other hand is what makes each product unique in that variable features are not found in every application. Variability has been shown to exist in time, and in space. (Pohl et al., 2005) has defined variability in time as *"the existence of difference versions of an artefact that are valid at different times"* and variability in space as *"the existence of an artefact in different shapes at the same time"*.

2.2.4 Feature-Oriented Software Development

Within this dissertation, we concentrate on the use of Feature-Oriented Software Development (FOSD) for developing SPLs (Apel and Kästner, 2009). FOSD is a set of different modelling tools, language techniques, centred around the concept of a Feature.

Feature

As FOSD aims at modularising and viewing a product as a collection of features, it is useful for us to have a clear understanding of what a *feature* actually is. In the past, several definitions of a feature have been proposed, and have been surveyed by (Apel and Kästner, 2009) (ordered from abstract to technical):

- *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems".* (Kang et al., 1990)

- *"a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained"* (Kang et al., 1998)
- *"a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept"* (Czarnecki and Eisenecker, 2000)
- *"a logical unit of behaviour specified by a set of functional and non-functional requirements"* (Bosch, 2000)
- *"a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"* (Chen et al., 2005)
- *"a product characteristic that is used in distinguishing programs within a family of related programs"* (Batory et al., 2004)
- *"a triplet, f = (R, W, S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification"* (Classen et al., 2008)
- *"an optional or incremental unit of functionality"* (Zave, 2003)
- *"an increment of program functionality"* (Batory, 2005)
- *"a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"* (Apel et al., 2008)

These definitions of feature though do not help explain exactly what a feature can be. Features can be a range of different items that are important to that product. Research into SPLs has predominately centred around features that contain source code, but features can encapsulate many more types of artefacts. Other types of considered artefacts include documentation (Rabiser et al., 2010). In this thesis, we mainly consider features as used for modularising software, and context. With software related features, Myers (1988) categorised features relating to capability into three areas:

- **functional.** (features that provide a service).
- **operational.** (features related to application interaction).
- **presentation.** (features related to how information is presented).

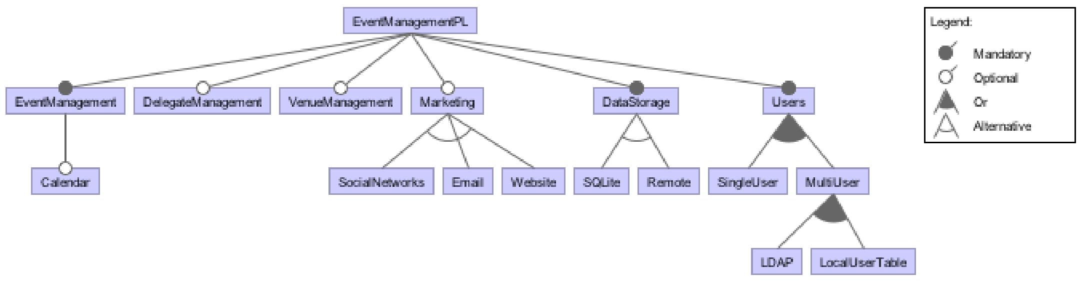


Figure 2.1: Example Feature Model

Feature Models

While there have been different modelling notations to model variability e.g. Orthogonal Variability Models (Pohl et al., 2006; Pohl et al., 2005) and Covamof (Sinnema et al., 2004), feature models have become the de facto technique, with an example in Figure 2.1.

The origins of the feature model can be traced back to Kang et al. (1990), who proposed the Feature-Oriented Domain Analysis method (FODA). FODA included the modelling concepts of aggregation/decomposition, generalisation/specialisation, and parameterisation. Feature models are modelled using hierarchical trees of features, with each node representing commonality and variability of its parent node.

Feature Relationships

As said before, feature models comprise of features and the relationships between them. There are different relationships that can be applied to features, including:

- **Mandatory**. A child feature is defined as mandatory when it is included in all products where its parent is also contained.
- **Optional**. A child feature defined as optional when it optionally can be included or excluded when its parent is contained in a product.
- **Or**. A set of feature children exhibit an or-relationship when at least one or more children are selected along with the parent of that set.
- **Alternative (XOR)**. A set of feature children exhibit an xor-relationship when only a single child can be selected when the parent is included in that product.

Relationships among features are not only expressed within the tree, but also by cross-tree constraints, written commonly using propositional formula. These cross-tree constraints typically apply feature inclusion or exclusion statements.

Propositional Formula

Feature models can be encoded in propositional logic, which then can be reasoned over for configurations (Batory, 2005). Propositional logic can be viewed as a logical representation of the feature model. By the use of logical connectives including $\vee$, $\wedge$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ with primitive variables (Boolean values), a formula can be determined to be either satisfiable, or non-satisfiable. Formula satisfiability is determined by whether the formula evaluates to true, given a set of variable assignments. Evaluating these formulas is usually carried out using SAT Solvers, BDD Solvers, Alloy, or SMV (Benavides et al., 2010). To use SAT solvers, the propositional formula has to be encoded into Conjunctive Normal Form (CNF). To evaluate a feature model then requires being mapped to propositional formula, normally using these following steps (Benavides et al., 2010; Czarnecki and Wasowski, 2007; Batory, 2005):

- Firstly, the feature model encoding is carried out by first assigning a variable to each feature.
- Secondly, each feature relationship within the feature model is encoded in to smaller formulas.
- Finally additional feature constraints added to the feature model are added to the formula.

Feature Model Extensions

Since the inception of feature models, a number of extensions have been proposed. First type of extensions includes the ability to add cardinalities to features (Czarnecki et al., 2005a; Czarnecki et al., 2004). In cardinality based feature models, cardinalities, similar to multiplicities in UML, can be added to features and groups. A cardinality attached to a feature states the number of instances that feature can be presented in a product, by stating the lower and upper bounds of a range. Group cardinalities on the other hand are used to denote the number of subfeatures that can be present in a product.

Other extensions to feature models include feature types. It was noted by Thum et al. (2011) that some features are not always used to distinguish program variations but used more to structure the feature model. It was proposed that features can be either:

- **Abstract**. A feature contains no implementation artefact, and is used primarily to help structure the feature model.
- **Concrete**. A feature that has at least one concrete implementation associated with it.

Binding attributes to features has also been a proposed extension (Benavides et al., 2010). Feature Models including attributes have been called *extended feature model*. Extended feature models give the ability to tag extra information to features. Feature attributes normally contain a name, a domain, and a value. This modelling extension is used later in this thesis for use with our context models.

2.2.5 Implementing Product Lines

Different software development technologies and paradigms have been used to engineer SPLs including Aspect-Oriented Software development and Service-Oriented Architecture. FOSD extends FODA by providing technologies to enable development and implementation of software systems in a feature oriented fashion.

Separation of Concerns

Separation of concerns is not a concept just regarding SPLs but a fundamental principle of software engineering (Apel, 2007). While a software concern can be seen as *any matter of interest in a software system* (Sutton and Rouvellou, 2004), because this variation in behaviour can affect different parts of the software, it is considered a *crosscutting concern*. Crosscutting concerns are non modular concerns, that are found scattered across multiple pieces of software modules (Bruntink et al., 2007). A concern can be any part of the system and be realised as a feature in SPLs, aspects in AOP, or class in OOP. It has been recognised that concerns cannot always be separated at the same time, known as the *tyranny of the dominant decomposition* (Tarr et al., 1999). Concerns that can not be modularised separately normally equate to *scattered, tangled, or replicated* in code (Apel, 2007). Code scattering occurs when concerns can be found to be implemented or scattered across multiple source code modules, for example classes in OOP. Code that is tangled can be found when more than a single concern are implemented together within a single source code module. In SPLs, features can be seen as different concerns of the system that can be separated.

The are two primary approaches to dealing with separation of concerns in source code including *annotations,* and *composition*.

Annotative and Compositional Approaches

Annotative and compositional approaches provide two different solutions to separating concerns. Annotative approaches focus around the use of virtual separation of concerns (Kästner and Apel, 2009). Examples of tools that support annotative approaches include the C preprocessor, and CIDE tool (Feigenspan et al., 2010). Using these tools, the developer can annotate parts of the source code that are part of each feature in the SPL. This approach can allow very fine grained adaptation including extra statements to methods, and parameter alterations in method declarations. Product derivation is carried out by negative variability. Using negative variability, parts of the system are removed based on which features are present in the product configuration (Voelter and Groher, 2007). This causes code to be removed from the final variant of the source code if its associated feature is not included in a product.

Compositional approaches on the other hand focus around physical separation of concerns. By physically separating code into multiple modules, these can be composed into different variants at configuration (Kästner et al., 2009). This approach normally makes use of positive variability because elements that are variable to the product are added to the base product (Voelter and Groher, 2007). Many languages supporting software composition exist include Aspect-Oriented Programming, Delta-Oriented Programming, and Feature-Oriented Programming. In this thesis, we concentrate on the use of Feature-Oriented Programming in implementing SPLs.

Feature-Oriented Programming

Feature-oriented programming (FOP) is a programming paradigm that modularises software according to *features* (Batory et al., 2004). Two prominent FOP languages include Jak for Java (Batory et al., 2004), and FeatureC++ for C++(Apel et al., 2005). In FOP, classes are implemented as standard classes within the base language. To implement a change to already defined classes within a feature, *refinements* are used.

Refinements contain the adaptation for a particular class. A refinement is declared using the keyword `refines` in the refinement class declaration. Within these refinements, extra class members, methods, and method extension can be added. Method extensions function by overriding the method being extended, adding extra instructions, and using the keyword `Super` to call the overridden method. The position of the `Super` keyword can then dictate when in execution should the extensions take place, in regards to the overridden method. In Listing 2.1, we depict an example from a program for events management. In lines 2-15, we declare the base version of the class `ManageEvents`, and a refinement for the SocialNetworks feature in lines 18-30. In the refinement, functionality related to sending twitter status updates for new events is added.

Listing 2.1: Adding support for support for tweeting new events (SocialNetworks)

```
 1  //Base implementation
 2  public class ManageEvents extends Activity {
 3    public void onCreate(Bundle savedInstanceState) {
 4      super.onCreate(savedInstanceState);
 5      setContentView(R.layout.manageeventlayout);
 6      ...
 7    }
 8    private void addNewEvent(Event e) {
 9      ...
10      events.add(e);
11      db.insertEvent(e);
12      updateMenuList();
13      ...
14    }
15  }
16
17  //Refinement for Feature SocialNetworks
18  refines class ManageEvents {
19    TwitterClient mTwitter;
20    public void onCreate(Bundle savedInstanceState) {
21      Super.onCreate(savedInstanceState);
22      mTwitter = new TwitterClient(getApplicationContext());
23    }
24    public void addNewEvent(Event e) {
25      Super.addNewEvent(e);
26      String t = getResources().getString(R.string.tweet_new_event) + e.name;
27      if (mTwitter.isLoggedIn())
28        new SendTweetTask().execute(t);
29    }
30  }
```

Two common techniques for software composition in FOP include *jampack*, and *mixin*:

- **Jampack** based composition involves composing classes into a single compound class of the base class and each of it refinements. Each of these compound classes then include a union of all member variables, and one method for each method refinement chained together. This method has advantages over mixin based composition through its ability to create smaller and more scalable codebase (Rosenmuller, 2011).
- **Mixin** based composition involves recreating an inheritance chain from classes in each feature. Using this approach, each refinement is added as an abstract class, with the bottom declaration a public class. Each of these refinements then extends the previous refinement class in the inheritance chain. This method has advantages over jampack based composition through the ability to keep feature boundaries and ease of refactoring changes in generated code back into feature increments (Batory et al., 2004).

In FOP, modules play a role in facilitating separation of concerns, known as *feature modules*. Feature modules encapsulate program refinements for that specific feature, which increments program functionality (Batory et al., 2004). In terms of their use, they are used via folders within an operating system to separate source code of different features. Feature modules are applied to a base program by a program generator, which then uses the modified program as the input for the following composition. This is commonly carried out using *superimposition* (Apel et al., 2009; Batory et al., 2004; Apel, Kastner and Lengauer, 2009)

Superimposition involves a given artefact being superimposed by their refinements, contained within feature modules. Apel and Lengauer (2008) proposed that to superimpose one program structure over another, each feature module is broken down into a tree, called a Feature Structure (FST) model. Within a FST model, different FST nodes represent different elements of a given artefact, for example package imports, classes, and methods. There are specifically two types of nodes, *nonterminal*, the inner nodes of the tree with recursively further nodes and *terminal*, the leaves in the tree. During composition, the base program and refinement FST trees are composed by recursively copying FSTNodes from the refinement tree that do not exist in the base tree. When composing two terminal nodes of the same identifier, different composition rules are needed to correct compose the two nodes. An example rule includes how to compose the bodies of two Java class methods.

This general purpose superimposition technique was used in the composition tool *FeatureHouse* (Apel, Kastner and Lengauer, 2009). Different languages can added to FeatureHouse by the use of a language BNF, and specific annotations marking different elements in the grammar as Feature terminals and non terminals. This allows for different levels of refinement granularity. FeatureHouse also has an altered syntax used for implementing class refinements. Firstly, unlike other FOP languages, the `refines` keyword is not used. Depending on the order of composition, when the classes are read, if a class declaration for the same type already exists, it is then assumed to be a refinement of the previously read class declaration. This then means the `refines` keyword is not required. Secondly, instead of the keyword `Super` for calling the overridden method in other FOP languages, FeatureHouse uses the keyword `original`.

2.3 Context-Aware Adaptive Applications

Software is being used in increasingly mobile environments. Because of this, conditions of the user, device and other factors are often changing. This change in conditions can cause problems with traditional static applications. For example, how does an application requiring data from an internet source deal with connection loss? These events can be handled by context-aware adaptive applications. Context-aware applications have been described by Daniele et al. (2009) as:

> *...intelligent applications that can monitor the users context and, in case of changes in this context, consequently adapt their behaviour in order to satisfy the users current needs or anticipate the users intentions.*

2.3.1 Context

Context-aware applications are driven by context, whereby, given a particular context, the application adapts itself. Because of this, it is useful for us to define and discuss what *context* actually is. This thesis will use the definition of context as given by Dey (2001):

> *Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves*

Context data can be acquired from a range of sources. Because of this, contexts sources can be broadly categorised as one of the following:

- **User**. These types of contexts represent different characteristics of users. This includes the person using the application directly, or could refer to other people of interest. Examples of user contexts can include user preferences, their current activity, if they are available etc.
- **Device**. This context category refers to the current environment in which the application is being executed. In our case, we refer to mobile devices. Examples of contexts can include changing values like battery level, network connectivity, storage space. We can also include more static characteristics of the system including what hardware exists, or what software is on the device e.g. the operating system version.
- **Environmental**. These contexts refer to external conditions of the device. As smart devices are typically equipped with different sensors like GPS/GLONASS, digital compass, and gyroscopes, different characteristics of the external environment can be determined. This type of information can be used for retrieving location based information for example.

This data is can be heterogenous in nature, which requires abstracting to more meaningful information. In this dissertation we consider how context is used to drive dynamic feature binding, whereby different features are only needed when a particular context is active.

2.3.2 Context Acquisition & Adaptation

The first part in context-ware adaptive applications includes the acquisition of context. Context acquisition can be described as the process of collecting data from various sources, and then interfering the context in which a particular entity is in. This data can be either purely in a raw format from electronic sensors e.g. GPS, accelerometers etc, or can also be inferred by the reasoning on a collection of contexts together.

The second part to context-aware adaptive applications is the adaptation. The software adaptation can be described as the change that the application makes when a particular context has been found to exist. McKinley et al. (2004) described two main types of software adaptation, *Parameter* and *Compositional*. Parameter adaptation relies on the modification of program variables. Compositional adaptation in comparison

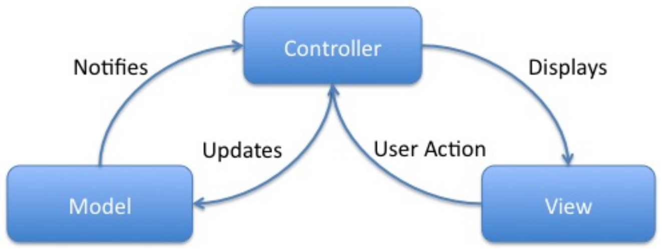


Figure 2.2: Model-View-Controller

alters the system logic to adapt itself. Context adaptation is defined at design-time, and then realised at runtime.

2.4 Graphical User Interfaces

An important aspect to software includes the Graphical User Interface (GUI). In this section, we focus on how graphical user interfaces are engineered and implemented. GUIs can be traced back to the Xerox Star Interface, which lead to the WIMP model. The WIMP model was a type of interface that comprised of windows, icons, menus, and pointers. This model of interface became dominant in the desktop era of computing. With the emergence of smart devices e.g the iPhone and Android, GUIs moved away from the WIMP model, and embraced touch. This included sets of different touch gestures including pinch and pull for enlarging/zooming on particular content. GUI development can be carried out from scratch, but is often carried out by the use of different software frameworks available to different platforms.

2.4.1 Model-View-Controller

In mobile and web application development, it is common to not have the entire application developed using a single language or technology. Because of the heterogeneous nature of mobile devices, it is suggested that GUIs should not be programmed along with business logic. This is to help alleviate some of the difficulties resulting from tailoring to suit multiple device screen sizes and device capabilities. Also, when developing user interfaces along with business logic, code is less maintainable and is harder to reuse when interface elements can be reused in multiple places of an application.

To improve this maintainability and reuse, different design patterns have been proposed, particularly the Model-View-Controller pattern.

The Model-View Controller (MVC) design pattern (Krasner and Pope, 1988) aims to improve development of graphical applications by separating information from its representation, and its interaction. It was originally introduced in Smalltalk-80, and since been used in web applications, and mobile application development. The MVC pattern is driven by three aspects including:

- Model: Encapsulation of data, and behaviour relating to its state.
- View: The representation of the model, visually displayed to the user.
- Controller: Handles events from the user and or device, causing view and model changes if necessary.

There are various benefits of using the MVC pattern. Particularly, UI design and implementation can be carried out separately to the software controlling it, allowing for different types of developers to develop different elements of an application easier, while improving software reuse and maintainability.

2.4.2 View Implementation

As said above, the View is the representation of the model, displayed to the user. In terms of a GUI, a View is not a single entity, but more formed of other views. Each view corresponds to different elements of the GUI. Examples of GUI elements include the following:

- **Textfields**: Elements that are designed for text input and output. These textfields are normally used within forms, for example a page for editing contact details within a contacts application.
- **Buttons**: Elements that are pushed/clicked on by the user, which normally correspond to some particular application behaviour. Buttons can be used with a range of appearances. Particularly, image buttons can be used as a way of making static images of a GUI clickable, for example clicking on a contact photo, which then can prompt the user if they wish to change the image for the contact.
- **Checkboxes**: Elements that are designed for binary on and off decisions. Within a group of checkboxes, any number of them can be manipulated, unlike radio buttons.

- **Radio Buttons**: Elements that are designed for the user to select a single option only. Radio buttons are used in groups of two or more, whereby each button corresponds to a single option.
- **Text Labels**: Elements designed to help label and explain different widgets and or groups of widgets. Labels can be used to help the user know what data is expected within input widgets. If many text fields are on a single form without labels, it can be hard for the user to know what data is expected in each. By using labels, the user can know what type of data is expected.
- **Layout Widgets**: Elements designed to aid the developer in the layout of other widgets, usually by their use as View containers. GUIs can be designed with exact positioning where each widget is placed in a static position, but these perform badly when a GUI is used on different screen dimensions. By using layout widgets, GUIs can be made to support a wide range of screen dimensions, through more relative placements of widgets.

When creating a View, by using these different elements together, a tree structure of nodes and branches is formed (Kramer et al., 2011). These trees can be therefore created by the aggregation of other small GUI trees, creating more complex GUIs, while allowing for re-use in multiple windows. The GUI tree for adding new blogs in the Wordpress Android application has been depicted in Figure 2.3. This tree has a number of different layout widgets for relative and linear positioning. There are also different text fields, labels and buttons in the GUI.

Document-Oriented GUIs

GUIs have often been implemented through code statements in general purpose languages e.g. C++, Java, Objective-C. Different elements of the GUI including visual properties and controls are created using program statements. This approach requires programming knowledge from the developer creating the GUI.

Within recent years, we have seen the emergence of GUI representation being implemented using documents instead of code (Draheim et al., 2006). Using this approach, GUI representation is implemented in a more declarative fashion, commonly in markup based languages (Kim and Lutteroth, 2009). Other names for this type of GUI declaration include GUI description languages (Limbourg et al., 2005). Examples of these languages include Mozilla XUL, QML used in QT, Microsoft XAML, Apple Nib,

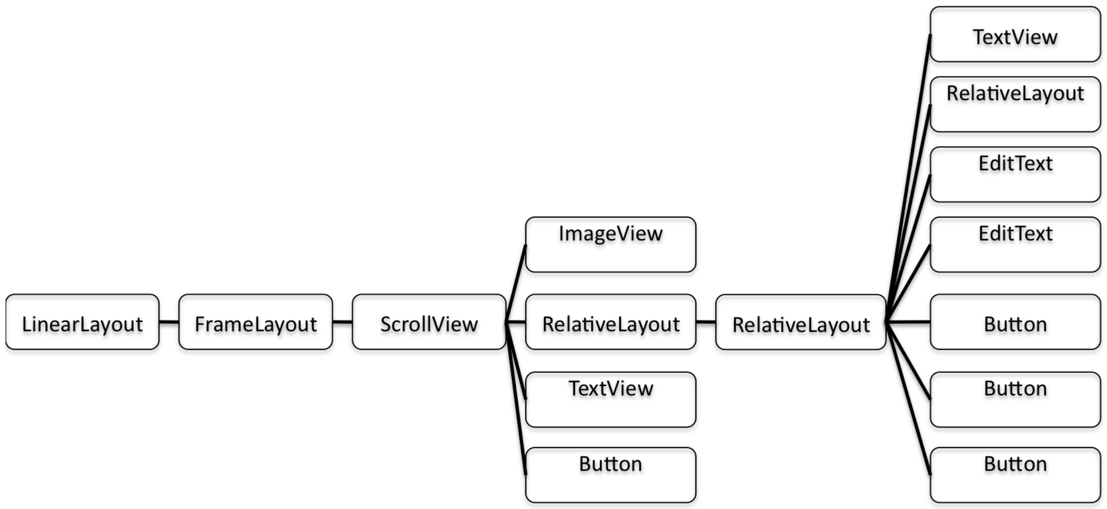


Figure 2.3: A GUI tree from the Android Wordpress Application

and Android XMLBlock. For performance reasons, during compilation, many of these formats are preprocessed into formats for use in the final application.

By using a document oriented approach, there are many advantages discussed in Draheim et al. (2006) including separation of concerns, compatability, editability, and non-universality and abstraction. Furthermore, with many of the platform developments, WYSIWYG editors are included, where a GUI can be built using drag and drop of widgets. This then means that the GUI can be previewed by the developer, without the need for compiling and running the application.

An excerpt of a GUI document for displaying blog posts in the Wordpress Android [1] application is shown in Listing 2.2. This document defines the layout for each row of a list of blog posts, and also is an example of how documents do not define an entire GUI, but can define smaller reusable fragments. This layout is a collection of TextViews that act as labels for displaying important summary information of a post, including title, and date in which it was published etc. The LinearLayouts are used as layout widgets, which help to position the TextViews.

GUI documents in different platforms, while achieving similar goals, can differ in what and how elements are expressed. In iOS development for example, Interface Builder files take a far more close to metal approach to what is expressed, whereby the GUI document contains an XML representation of the runtime object graph. Because of this, to avoid errors, the developer should keep the graphical tools of the SDK for editing these documents. Other platforms do not always take this approach, and

[1] http://android.wordpress.org/development/

instead use GUI documents that are far easier to edit by hand.

Listing 2.2: An Android GUI document

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/row_post_root"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    style="@style/WordPressListRowBackground"
    ....>
    <TextView
        android:id="@+id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#464646"
        android:textSize="18sp" />
    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        ....>
        <TextView
            android:id="@+id/date"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#777777"
            android:textSize="12sp" />
        ...
    </LinearLayout>
</LinearLayout>
```

GUI documents are primarily designed for defining the visual representation of the GUI. This means that other aspects of the GUI including event listeners need to be defined within a view controller, or similar. In these controllers, GUI documents are instantiated in code, whereby GUI objects are created based on the documents. These objects are then displayed to the user and can be manipulated in code.

2.4.3 Dynamic GUIs

So far, we have discussed development of predominantly static GUIs. Nowadays though, as discussed earlier, the conditions in which the user and the application run is often changing with mobile devices. This has given a rise to more dynamic GUIs that can be adapted at runtime.

When discussing non static GUIs, two popular approaches have been proposed, *Adaptable* and *Adaptive* (Stuerzlinger et al., 2006). When a GUI is described as being adaptable, it means that the GUI can be altered by the user at runtime to suit their own needs. Alternatively, a GUI that is adaptive is not altered by the user, but by the system automatically based on different events or usage patterns. Many examples of adaptive GUIs exist, for example, the Eclipse[2] development environment can be considered as having an adaptive and adaptable GUI, due to additional buttons and controls being added to a users environment on installation of additional plugins. This GUI also allows the user to alter certain aspects of the GUI. In this dissertation our focus is centred around adaptive applications only.

Types of Changes

Using Adaptive GUIs (AGUI), the GUI can be subject to a number of changes. Here we attempt to look at the dynamic changes used in previous work to help understand the sort of changes we expect in a dynamic GUI, which will be used in the rest of this thesis:

- **Alteration in layout based on visible widgets.** This type of alteration can be based on the inclusion or deduction of widgets from a given layout. Depending on the condition, it may be right to add or remove different widgets on the screen, to help the user. An example of this includes removing buttons for unused functionality in a program, making the software more well suited for a usage pattern (Findlater and McGrenere, 2010).
- **Alteration in widget visual properties.** This alteration is primarily aimed at the alteration of widgets already on the screen. This can mean altering size and dimensions, or just give visual appearance (background image/colour) of the widget. An example of this includes indicating to the user that sound output is now muted by the alteration of the widget showing system sound volume (Paymans et al., 2004).
- **Alteration in widget behaviour.** Widget behaviour alteration encompasses changes in how the widget acts, or logic the widget invokes when clicked/pushed.

Usability of Adaptive GUIs

When considering GUIs that can be adaptive, one must always consider the usability effects. Several studies have considered the effects of adaptive GUIs on usability. In

[2]http://www.eclipse.org/

Paymans et al. (2004), the authors acknowledge that AGUIs can cause users to feel a lack of control, due to the interface being unpredictable and incomprehensible. As an experiment, it was considered that helping to build greater mental models of the system would increase the usability of the system. This study was carried out with 17 students from the Utrecht University, of which 8 were male, and 9 were female. The authors measured learnability and ease of use while simulated context changes were carried out using the tasks of watching video streams. Before and after watching the video streams each user had to complete a usability questionnaire. The study found that an improved mental model was not always needed. It was suggested that depending on the domain, to achieve ease of use, it may not be necessary to have an understanding of the system's design.

Later studies including Lavie and Meyer (2010) examined the usability effects of adaptive GUIs within the context of in-vehicle systems. Within this study, four different factors were observed, including different tasks, routine and non-routine situations, age groups, and different levels of adaptivity. Different age groups were handle via two separate experiments, one for younger users using 64 participants with an average age of 25.7, and for older users using 24 participants with an average age of 58.6. The users were asked to carried out various takes, both routine and non routine, including reading a SMS message and sending a reply, reading an email message, reading news updates, and changing cd. It was found that intermediate level of adaptivity may be more beneficial to the user and that adaptivity does not benefit every condition. Fully adaptive systems were found to be more beneficial on routine user tasks.

2.5 Summary

In this chapter, the foundations of the subject areas in this thesis have been introduced. Firstly we introduced Software Product Lines as a software engineering approach. Following this, context-aware adaptive applications were introduced. Finally, we introduced foundations in to GUIs, including adaptive GUIs.

In the next chapter, works that are closely related to our approach are surveyed and discussed. We firstly explain how SPLs can handle dynamic behaviour. Next, we discuss how context has been used in previous work, by adaptive applications, and dynamic SPLs. Also, variability of GUIs is considered, both statically and dynamically. This will allow the contributions of this thesis to be compared, and positioned concretely. Following this, our contribution can be described and discussed later in the thesis.

3 Dynamic SPLs and Adaptive GUIs

Contents

3.1 Introduction

In this chapter we discuss the state-of-the-art regarding research in Dynamic Software Product Lines (DSPL), Graphical User Interface (GUI) Adaptation, and Context-Awareness. In Section 3.2, we introduce Dynamic Software Product Lines, discussing the properties of a DSPL, and surveying many of the proposed approaches. Techniques to handle static and dynamic adaptation of the GUI is then surveyed in Section 3.3. Section 3.4 discusses different approaches to context-awareness, and their involvement in DSPLs. We then revisit and outline our research goals in Section 3.5, and conclude the chapter in Section 3.6.

3.2 Dynamic Software Product Lines

Until the last few years, SPLs were traditionally used for the creation of static software systems. These products were normally statically bound before the software is executed, and therefore cannot adapt at runtime. Lately the need for dynamic binding and the ability to alter a programs configuration at runtime has become a reality. Dynamic SPLs (DSPL) have been proposed as SPLs that bind variation points at runtime, which can exhibit the following properties (Hallsteinsen et al., 2008):

- dynamic variability: configuration and binding at runtime
- changes binding several times during its lifetime
- variation points change during runtime: variation point addition (by extending one variation point)
- deals with unexpected changes (in some limited way)
- deals with changes by users, such as functional or quality requirements
- context-awareness (optional)
- automatic decision making (optional)
- individual environment/context situation instead of a ”market”

DSPLs have shown to bring benefits in ubiquitous environments, particularly in the mobile market, where device capabilities are highly heterogeneous (Kaviani et al., 2008). At the heart of a DSPL lies its ever changing configuration.

3.2.1 Configuring a DSPL

DSPLs like SPLs get configured, but not every feature in DSPL may require dynamic binding. While some features binding needs to be carried out at runtime, some feature binding can be carried out at compile time. This means the configuration of the DSPL is not always configured within a single step or process. This can be viewed as *staged configuration* or *SPL specialisation* (Czarnecki et al., 2005b). By having a staged configuration, complete product configuration is not carried out within a single step, but more across multiple steps, or stages. With each step, the variability of a

SPL is reduced, also known as a specialisation, until the SPL is fully specialised with the configuration being complete. In static SPLs, staged configurations can be carried out by multiple actors, each only configuring a part of the SPL. For DSPLs, the runtime contexts that are acquired at runtime can be considered the actors of the final configuration/specialisation. In DSPLs, it has been shown by Rosenmüller et al. (2011a) that carrying out a stage configuration by aggregating features together statically can improve not only runtime performance, but also reduce compositional overhead needed for each dynamic bound feature. Unlike static SPLs, a DSPL can be reconfigured, binding new features in the place of others, in a continuous process.

3.2.2 Implementation Approaches

Several implementation approaches have been proposed for DSPLs, including language extensions, and components and services. Here, we aim to review the different approaches taken for implementing DSPLs, firstly looking at language support through language extensions, before looking at the use of components and services.

Language Support

Different approaches in the form of language support and extensions has been an area of interest in the area of dynamic software product lines. These language approaches attempt to consider the problem of cross cutting logic, and how logic can be added or removed at runtime. As discussed in the earlier chapter, Feature Oriented Programming (FOP) languages like FeatureC++ (Rosenmüller et al., 2008) have been proposed. In FeatureC++, logic within the dynamic feature modules is modularised using the decorator pattern. Using this pattern, decorators wrap classes at runtime to alter the behaviour of an application. For each feature, a class is generated, to which on feature binding, all decorators needed are loaded and the base classes are wrapped. This method was found to help reduce functional overhead, caused by unused features within a product variant. Later, Rosenmüller et al. (2011a) applied a generative approach to FeatureC++, whereby, in situations that multiple features are used together, it is beneficial to statically compose several features together into composite features, named Dynamic Binding Units (DBU). This is handled by manually grouping features into DBUs, to which the feature model is then automatically reduced and DBUs are checked to avoid invalid compositions. This method was found to help balance the need for reducing functional overhead without merely replacing it with compositional overhead.

Another programming paradigm to deal with feature style crosscutting concerns includes Delta-Oriented Programming (DOP), particularly DeltaJava (Schaefer et al., 2010) and its dynamic form, Dynamic Delta Oriented Programming (DDOP) (Damiani and Schaefer, 2011). In DOP, feature refinements are implemented in *Delta Modules*. A delta module though similar to feature modules in that it can increment a base program functionality, it can also remove functionality. Other interesting features of DDOP is the promise of altering application state, by applying stack changes when a delta is applied. Delta Modules are implemented as single files, instead of each class refinement being a class file as in FOP. Currently, only semantics of DDOP have been proposed, and a working prototype is yet to be proposed.

Other language extensions that while are not explicitly defined as being used in implementing DSPLs includes Context-Oriented Programming languages. These languages crosscut programs based on different contexts by the use of layers (Hirschfeld et al., 2008). Layers can be defined with OOP classes, or in their own self contained layer (Appeltauer et al., 2010). At runtime, layers are then activated or deactivated. Java based COP languages have primarily been implemented on top of AspectJ. COP languages though only normally consider dynamic software composition, which has been found in FOP (Rosenmüller et al., 2011a) work to be very useful in lowering compositional overhead when a particular adaptation is required statically in a program. While this programming paradigm is largely proposed to differ from FOP by its concentration purely on runtime adaptation, it is still possible to use this approach, along with static composition using other tools, for example, Featurehouse (Apel, Kastner and Lengauer, 2009).

Components and Services

Components and services have also been proposed as a method to producing DSPLs. Service Oriented Architectures (SOA) recently has been a popular domain for DSPL research. Parra et al. (2009) first proposed CAPucine, a usable method for using SOA and Service Component Architecture (SCA) for creating a DSPL. This methodology also incorporated the use of context to drive its derivation process, which is discussed later. For realisation, the authors use the FraSCAti SCA platform (consortium, n.d.). The approach was broken down into two phases, the *initial*, and the *iterative* phase. The initial phase encompasses the structural and business modelling, behaviour modelling, and implementation. Implementation of the DSPL of this phase is carried out by the composition of feature assets, and a number of transformations from the application metamodel to source code. For model transformation and code generation, Kermeta for generating Java and SCA elements from the application model, with the Acceleo language used for generating Java and SCA source code. Within the iterative phase, FPath and FScript domain specific languages are used for handling the runtime adaptation of the system. FPath is used for stating what components will be adapted at runtime, with FScript being used for unbinding, and binding the new components to the system in using transactions, ensuring a consistent state.

A SOA approach has been proposed within the domain of mobile computing (Marinho et al., 2010). The proposed approach was applied to the development of context-aware mobile guides. This methodology was proposed to be carried out over three stages, two of which were within the domain engineering stage. Firstly, all meta components required for the dynamically adapting the system were analysed in stage 1. Then in stage 2, all commonality and variability for the mobile guide domain was analysed. Finally in stage 3, the domain requirements for the Great Tour mobile guide was analysed. Transitions between stages was carried out by considering what features from the previous stage are needed, and then moving them to a more specialised feature model. For feature and context modelling, UbiFex (Fernandes et al., 2011) was used, which is discussed later. Different problems encountered in the process were presented, grouped into the areas of domain, theory, project management, and tools. Interestingly, while the main context middleware was created in the first two stages of the development, all components and services used in the final application were created only for that application. It was further claimed that the components could be rewritten to a more reusable solution, but no method was proposed or cited.

A further SOA adaptation approach and platform was proposed by Gomaa and Hashimoto (2011). This approach built on from a previously proposed evolutionary process model for software product lines. The platform proposed builds on top of the SASSY framework (Malek et al., 2009). The platform takes the form of a 3 tier architecture including *Goal management*, *change management*, and *component control*. The goal management tier primarily deals with SPL feature selection when feature binding needs to change. The change management tier has two service, gauge service which checks if a configuration change is needed, and change management service which maintains the mapping between features and components, and determines what components need adding/removing. Lastly, the component control tier handles two services including a monitoring service for monitoring the running system, and adaptation services for the system adaptation.

Quality of Service (QoS) in SOA based SPLs includes the work of Lee et al. (2012). This work work proposes a QoS specification that is used to manage system quality in the monitoring and negotiation process. A QoS framework is used which includes systems for brokerage, reputation and service rating, and monitoring. The brokerage system includes the ability to compare the proposed renegotiation with other providers to. The reputation system is used by consumers to rate the quality of different service providers. Finally, the monitoring system monitors the quality of negotiated services, checking for SLA violations and runtime failures. For reconfiguration, a reconfigurator is proposed to trigger reconfigurations based on the current status of service.

Other service areas where DSPLs have been used include Web Services (Alferez and Pelechano, 2011). During the Domain Engineering stage, several models are created including a Feature model, a Compositional model, a Weaving model, a Context model, and a feature model for measure instruments. The compositional model makes use of a UML Activity diagram, as a method of describing the business logic and the workflow of the system. Secondly, the weaving models are used for mapping features within the feature model to the different elements in the composition models. Thirdly, the context model is carried out using Web Ontology Language (OWL), enabling context sharing and reasoning. Lastly, the feature model for measure instruments is created for enabling reuse of instruments for monitoring various service compositions. For the Application Engineering stage, an initial configuration of all the models created in the domain activity is defined, including the product specific context rules and their system resolutions. For runtime reconfiguration, the authors proposes an extension to their Model-based Reconfiguration Engine (MoRE) (Cetina et al., 2009) for Web Services which follows a MAPE-K loop (IBM, 2003) approach.

3.2.3 Summary of DSPLs

Currently, DSPLs have became a useful approach for dealing with static and dynamic variability in software systems. For this, different approaches have been proposed including language extensions and component/service based systems. We can see those that why these approaches help deal with variability in program logic, they fail to consider the GUI explicitly. While GUI changes can be defined in program logic, these logic extensions are only executed when the base methods are invoked. Therefore, GUI adaptations may or may not be carried out after a reconfiguration. It is for this reason, we explore different approaches to handling static and dynamic GUI variability in the following section to review the current approaches adopted outside of the DSPL spectrum.

3.3 Graphical User Interface Adaptation

Software is developed for various situations, whether it is for some backend functionality, or for the end user. Much of the software developed for mobile devices today can be described as end user applications, whereby the application is mixed with a rich GUI that the user interacts with. As we have discussed how general software variability is handled, equally, in many cases, a GUI can be crosscut by different features of a system, and should be handled in a reusable way. While we have surveyed existing approaches to DSPLs, they all focus primarily on logic adaptations, and do not address how variability in the GUI are handled. It is therefore in this section, we survey approaches proposed for dealing with variability that is static and dynamic regarding GUIs.

3.3.1 Design Time Adaptation

In this subsection, we survey approaches that deal with design time adaptation of GUIs within a product. These adaptations are applied statically during the development/derivation of the product, and can therefore be considered static variability. While different ad hoc solutions can be used for dealing with GUI personalisation, here we only consider approaches that propose a well defined approach or technology.

Handling static variability and supporting reuse of GUIs in a systematic way can be traced back to Schlee (2002) who adopted generative programming techniques and applied them for generating GUIs. This method was carried out by modelling abstract parts of the GUI within a conventional feature model. To derive a GUI variant, the authors propose to use the tool, ANGIE-Based GUI Generator (ABA), which takes an xml specification of the GUI, generated from a dialog-based graphical-interactive DSL.

SPLs using Model-Based UI Development (MBUID) have been a particularly popular method for handling UI variability. MBUID Hauptmann (2010) firstly applied this approach to web applications. In this approach the author proposed a methodology for handling UI within domain and application engineering. In the domain engineering, two types of artefacts are created, a feature model, and domain artefacts. Domain artefacts created include the application core, which is expressed in models for platform independence including a data model & operations, and an Abstract User Interface (AUI) models. Each AUI model can be seen as a combination of task model, and abstract user interfaces from model-based UI development. AUI models are structured in tree hierarchies of different nodes, along with relationships between them using temporal operators. Next, elements in the AUI model are mapped to feature in the feature model, and linked with the application core using a linking element. In the application engineering, the processes of product configuration and derivation is carried out. Product configuration involves the picking of what features should be added to a specific product. Product derivation produces a product model, made up of a UML based Web Engineering (UWE) content model, a UWE user model, a UWE process model, and AUI model. These are then transformed using a semi-automatic stepwise approach using model-based user interface development. This methodology was then applied to web applications creating JSP pages and some limited form of business logic of the application. This work was then described by Pleuss et al. (2012a). The authors also considered the need for manual customisation with automatic UI variant generation. This problem is caused by the need for manual modifications by of an applications UI after product derivation by customers of the system. The authors describe different aspects of MBUID commonly requiring customisation, how these different aspects can be customised.

The problem of creating UI for different devices and appliances has been approaches by Nicols (2006), who proposed a method for automated UI generation. Generation was carried out in two stages. The first stage took an appliance specification, written in a DSL, and produces an abstract user interface (AUI). Following this, interface modifications are carried out to ensure consistency among other interfaces created for that device. These modifications can be functional, or structural. From here, a concrete UI (CUI) is created, using platform specific UI objects. This is carried out by traversing the AUI tree, applying CUI rules. Lastly, the CUI is modified for consistency by the use of rules.

Other work concentrating on GUIs within an SPL considers how to re-engineer configurators (Boucher et al., 2012). In this work, the authors present challenges regarding the reverse engineering of existing configurators analysing GUI, webpage source, and code base to extract variability information. It is proposed that variability information can be extracted by searching for variability and constraint patterns in the GUI, with a few patterns already supported. a TVL model is generated after the user is satisfied with the extracted data in a post-processing step.Additionally, the challenge regarding forward engineering and generating a tailored GUI and codebase is discussed. It is suggested this can be handled by

Summary of static GUI variability

We can seen that design time adaptation of the GUI, with systematic reuse has been of interest. These methods have considered how GUIs can be customised, using reusable adaptations. Different methods have been show to have been used, including generative methods, and model based methods.

Having said this, considering they only tackle the need for design time adaptation, they are not adequate solutions for DSPLs. This is primarily because they do not support runtime GUI adaptation. When considering GUI adaptation in a DSPL, adaptation should be able to be applied both at design time, and at runtime. Next, we consider how dynamic GUI variability has been addressed in previous work.

3.3.2 Dynamic Variability

Dealing with dynamic variability have yet to be addressed using SPL approaches. Currently approaches have been predominantly from the adaptive GUI community, to which we will survey some of the most prominent works.

Adaptive GUIs have been addressed using a middleware approach (David et al., 2011). The authors make use of the Context-Oriented Programming language ContextJ (Appeltauer et al., 2009) for implementing UI changes in the application. When evaluating the middleware against developing the application using standard Android, the proposed solution allowed for a reduction in development time, testing time, while also being implementable with less lines of code.

A model-based GUI approach to context-aware AGUIs was proposed by Hanumansetty (2004). Proposed is a framework for handling web applications involving context processing and interface adaptation off device. Client side contexts are collected from the device and are sent to a context server, where with sensors and other system contexts are aggregated, and interpreted. Context events, are then sent to the business components and interface server. Based on different task model adaptation rules specified, the task model is regenerated for the GUI. Once the task model is updated, the abstract UI is generated using the dialogue model, and then the concrete

UI is generated using the presentation model. Generated UIs using approach were XHTML documents.

A stepwise composition approach to adaptive GUIs has been proposed by Savidis and Stephanidis (2010). This approach helps bring adaptivity to previous static GUIs using a refactoring process. This approach was proposed to handle interfaces implemented in OOP languages including C++. The approach is broken down into three distinct steps. In the first stage, user requirements are analysed, with roles and requirements identified. Then, the interface profiles is modelled, expressing variations of the user-interface behaviour. This step ends with adaptation decision logic being identified, where context events and their adaptation rule are specified. In the second stage, adaptation alternatives are encapsulated, by use of a general superclass, and each alternative component being a subclass. Dynamic replacement is handled by the termination of a component, and the activation by a substitute. During the component replacement, state is parsed from the original component to the constructor of the substitute.

Model-driven software engineering (MDSE) approaches have been proposed for dynamic GUI adaptation (Rodríguez-Gracia et al., 2012). GUI specifications are described using architectural models, which are altered runtime due to context changes. Adaptation is carried out by the use of model to model transformations, carried out at runtime, using rule models. Other MDSE approaches include the work of Criado et al. (2010). In this approach is broken down into two distinct processes. The first process is a model-to-model transformation of interface architecture models. Each interface architecture model is made up of abstract GUI components, and the M2M process is driven by user or application events, producing abstract GUI models. Model transformations were implemented using ATLAS Transformation Language (ATL), a Domain Specific Language for describing model transformations. Transformations are handled in two stages, first producing a intermediate model by taking the original model and the system event. The final transformation involves executing specific component actions with the state of the original model. In the second process of the approach, the abstract GUI models are then instantiated by getting the appropriate widgets from a widget repository.

A toolkit for handling transparent GUI migration and adaptation was proposed by Grolaux (2007). This approach was designed to be used with the Mozart Programming System[1], based on the Oz language providing declarative, object oriented, and

[1] http://www.mozart-oz.org/

constraint programming. Dynamic adaptation of the GUI is carried by each widget having different representations, that can be switched at runtime depending on a given context. Each of these representations is supported by individual renderers, which in turn represent a variant of that widget. These renderers can be distributed and transferred between applications, and even devices over an network interface. Granularity of adaptation supported ranges from the entire screen, to single widgets, and to an arbitrary pixel area.

Other approaches proposed include rule-based approaches (Paskalev, 2009). Using this approach, GUI descriptions are stored within database tables, that are loaded and transformed into object hierarchies. These objects are then converted to CLIPS facts for a engine proposed in previous work (Paskalev and Nikolov, 2004). The CLIPS facts are script files that describe different GUI parameters. Two processes are handled by the specified rules, first reconfiguring the UI for a given event, and then adapting the UI. Currently, only rules for hiding an removing certain GUI elements have been realised.

The need for scaling the GUI on mobile devices based on variable screen sizes was presented by Behan and Krejcar (2012). To help graphical scaling, the author propose the use of Scalable Vector Graphics (SVG). By using SVG graphics for different UI elements on the screen, these can be scaled to suit the screen size more easily, while retaining image quality. To use vector graphics instead of the default raster graphics, the authors propose to override the standard widget drawing methods. These override methods then instead translates the vector graphics to a drawable that can be used by the widget, scaling to suit the given display size.

Summary of dynamic GUI variability

Many approaches offer solutions to supporting dynamic GUI variability. Of these solutions, source code and MDSE have been the two most popular methods. While the need for dealing with device variability has been approached, this only considered scaling the same GUI, and not altering what UI elements are on the screen. Some of the approaches allow for context rules to be defined, causing a reconfiguration on context change. While these approaches provide solutions to dealing with dynamic GUI variability, they do not provide a method of handling static GUI variability.

3.3.3 Mixed Variability

Next, we consider work that considers GUI variability that can be handled statically and dynamically. In this category, we consider *Plastic User Interfaces* (PUI) (Calvary et al., 2001). Calvary et al. recognised both the need for dealing with both variability in devices in which an application many run on, and the need to deal with environmental changes of the device. The level of plasticity of a user interface is defined as its ability to adapt to different contexts, whereby the more contexts the UI can adapt to, the higher the plasticity. PUIs have been proposed as method for handling both adaptive, and adaptable UIs. A reference framework for plasticity was proposed, adopted from model-based UI development. Lastly a tool named ARTStudio was developed for developing all models except the environment and evolution models.

Plasticity has been proposed to be modelled as finite state machines, using mealy machines (Collignon et al., 2008) to handle UI resizing operations. Each state in a mealy machine is a resizing operation, with each transition being composed of source and destination of the GUI. This approach then uses UsiXML, a language for defining the final GUI, and is expanded for adaptivity and multi-presentation UIs. The specification language is expanded by adding different concepts including a set of *plasticitydomain*. This set of plasticity domains includes the different different conditions that are required for a particular variant of the UI. These conditions include characteristics of the platform, user and/or environment. Each plasticity domain is mapped to a specific GUI representation using inter-model relationships.

Coutaz et al. (2007) proposed an approach using MDE and SOA for developing PUIs. This approach is based on two principles. The first principle is that an interactive system is a graph of models. These models while developed at design time, still should be available at runtime, and linked by mappings. Concrete UI interactors should be mapped to the platform input and output devices, whereas task and concepts are mapped to the functional core entities. Transformations and mappings are models too, which are expressed in ATL. The second principle is that close-adaptiveness and open-adaptiveness cooperate. This is based around the need for self-contained and sometimes runtime extendable adaptation. Context use and UI adaptation is handled by services in the Distribution-Migration-Remolding middleware. Context observers gather contextual information that is processed by the situation synthesizer. New situations are then sent to the evolution engine to start adaptation. Adaptation can target either a section of, or the whole UI, using a mix of specifications, defined by the developer. The evolution engine then provides the configurator with what components need to be replaced and/or suppressed. These components then are then retrieved from the storage space if needed. Further work by Vanderdonckt et al. (2008) promoted a third principle of the keeping the user in the loop. This principle is based on the proposition that the user should remain in control of the UI, even if the UI operation is automatic. To support this, three types of adaptation are suggested including *automated, semi-automated,* and *manual transformations*. Designers and users can perform manual and semi-automated transformations. In semi-automated transformations, the designer can adjust the transformation target models at runtime.

Summary of mixed GUI variability

In this section we described approaches that propose solutions to dealing with static and dynamic GUI variability. Most of the approaches use context-awareness to drive GUI adaptation, and use MDE for implementation. These approaches proposed mostly fall in the category of PUIs. These approaches provide an interesting solution to dealing with different screen sizes, allowing the UI to better fit the screen. They do however have shortcomings. First, these approaches appear to concentrate on just screen sizes and resolution. This means that other runtime factors including location, cannot affect the presentation. We believe that UI changes should not be confined to particular contextual changes. Furthermore, there is no indication that other non presentation UI changes can be made for example behaviour. Particularly, when handling users from different regions of the world and platforms, different user gestures might be required. It is possible that because PUIs appear to have not been used in modern mobile platforms, different gestures are not required, or supported. However, for many modern mobile apps, gesture support is an important issue as discussed in the following chapter.

Other shortcomings of these approaches is that lack of a refinement or feature-oriented approach to handling these adaptations. Adaptations are handled as whole configurations, which do not allow for the same amount of reuse found using SPL approaches.

3.3.4 Summary

In Table 3.1, we summarise the different related work in the area of GUI adaptation in chronological order. We consider different characteristics to evaluate each of the

Reference	Representation			Adaptation		SPL Variability	Context-Awareness
	Code	Model Based	DOG/GDL	Design-Time	Runtime		
Calvary et al. (2001)	-	x	-	x	x	-	-
Schlee (2002)	-	-	x	x	-	x	N/A
Schlee and Vanderdonckt (2004)	-	-	x	x	-	x	N/A
Hanumansetty (2004)	-	x	-	-	x	-	x
Nicols (2006)	-	x	-	x	-	-	N/A
Coutaz et al. (2007)	-	-	x	x	-	x	N/A
Grolaux (2007)	x	-	-	-	x	-	-
Collignon et al. (2008)	-	-	x	x	-	x	N/A
Vanderdonckt et al. (2008)	-	-	x	x	-	x	N/A
Paskalev (2009)	-	-	-	-	-	-	-
Criado et al. (2010)	-	x	-	-	x	-	-
Hauptmann (2010)	-	x	-	x	-	x	N/A
Savidis and Stephanidis (2010)	x	-	-	-	x	-	-
David et al. (2011)	x	-	-	-	x	-	x
Behan and Krejcar (2012)	-	-	-	-	-	-	-
Boucher et al. (2012)	-	-	x	x	-	-	N/A
Pleuss et al. (2012a)	-	x	-	x	-	x	N/A
Rodríguez-Gracia et al. (2012)	-	x	-	-	x	-	-

Table 3.1: Summary of GUI adaptation approaches

works. These characteristics include variability representation, including source code, model based, or using Document-Oriented GUIs/GUI description languages. We also consider whether the adaptations can be used at compile time or runtime. Lastly, we consider if the variability is handled using SPL techniques, and whether context-awareness is supported.

We see that while GUI variability has been considered within SPLs statically and by non SPL approaches, but then dynamic variability of GUIs has only been considered within the adaptive GUI work. This means that the static and dynamic adaptations have to be realised using independent methods and processes, which makes reuse increasing difficult. Also, if the binding time of the feature is changed, this would require a reimplementation of that particular feature.

Since previous work helps show how crosscutting features can apply to GUIs as well as program logic, this then should be capable of being handled at runtime. GUI changes in previous approaches have to be carried out by logic increments. This not only avoids the ability to use modern document-oriented GUIs, but it also restricts changes to when a particular method is called. This is because in previous approaches, concentration was on logic changes. Unless an adapted method is invoked, that adaptation is not called. GUI adaptations may need to be carried out immediately, and therefore needs to be invoked on feature-reconfiguration.

3.4 Context-Awareness

Context-Awareness as introduced in the previous chapter has, with many general frameworks and libraries being proposed. In this section, first we review general purpose context management and modelling approaches. Next, we discuss how context has been used in regards to DSPLs. Finally we end this section with a summary of the approaches proposed, any gaps in knowledge, and issues that should be addressed.

3.4.1 General Purpose Approaches

Approaches for aiding development of context-aware applications goes back many years. One of the earliest and notable works includes the Context Toolkit (Dey et al., 2001). The Context Toolkit is a framework for rapid context-aware application prototyping. This framework is developed around components including context widgets for context information retrieval, interpreters for abstracting context information, aggregators for combining multiple related context information, services for executing actions e.g altering activator states, and discoverers which maintain a registry of framework existing capabilities.

Other notable context management frameworks include COSMOS (Conan et al., 2007). At the heart of COSMOS are *context nodes*, which are singular monitoring units, and uses software components and design patterns within its architecture. Context nodes contain several properties include whether they are passive/active, to observe/notify, and if they are blocking/not. Context aggregation to obtain high level context information is handled via *context policies*. Context policies contain a hierarchy of context nodes, allowing context nodes to be used across multiple context policies, sharing context information.

Different modelling approaches have been proposed for modelling context. Of the most notable work include the *Context Modelling Language* (CML) (Henricksen and Indulska, 2006). The CML is an extension of modelling concepts from Object-Role Modelling (ORM), due to its high formality and expressiveness. A mapping from ORM to a relational representation of CML fact types is used for different runtime context management tasks including persisting to a database, application querying, and constraint enforcement. In addition, a preference model is suggested as a method of assisting the decision-making process by allowing the user to set his or her own requirements. Two programming models are also proposed including *Branching*, for use when deciding over competing alternative context choices; and *Triggering*, which invokes different actions on context change.

Feature-Oriented Domain Analysis (FODA) has also inspired context modelling with the proposed *Context-Oriented Domain Analysis* (CODA) (Desmet et al., 2007). It is possible to see this a method for modelling DSPLs, except there are some fundamental differences. Firstly within this modelling notion, only context can be an actor in the configuration of the product, whereas in DSPLs, context and people in different stages of the stage configuration can be actors.

3.4.2 Context and DSPLs

While context-awareness has been used in adaptive applications, Lee and Kang (2006) proposed to use SPL principles in making adaptive applications by making SPLs reconfigurable after deployment. Since this work, there have been numerous modelling techniques used for the design, modelling, and implementation of context-aware

DSPLs.

Because context-aware systems rely on the use of context to determine its run-time behaviour, it has been highlighted in Fernandes et al. (2011) that context should be modelled along side feature modelling. Fernandes *et al.* proposed an extension to Odyssey-FEX feature modelling notation, named UbiFEX to represent context information and context rule specification. Within this notation context information is modelled using a context feature model. A context feature model is made up of two feature types, *context entity* and *context information* features. A context entity feature represents a specific domain context entity e.g. user, and mobile device, which has a name, and a description property. Context information features store the raw data that is needed to describe a context entity. These features require a name, description, whether they are static or dynamic, their type e.g. string, integer etc, initial value, and source. Following this, context definitions need to be defined according to a BNF notation. Lastly context rules are defined using the context feature, and the feature that needs to be activated separated with an implication operator. This modelling notation has been used in a mobile DSPL case study by Marinho et al. (2010).

Pure feature modelling has been used before for context models. Particularly, Acher et al. (2009) proposed to use multiple feature models to model context and its adaptation for dynamic adaptive systems. The context model is modelled as a single feature model using the FM hierarchy structure for context granularity. Context rules that cause feature selection changes are then expressed in cross model constraints, which are written the same as inter-model features constraints. Then, both the system and context feature models are aggregated into a larger feature model. This enables the inter-model constraints to function as constraints in singular feature models.

Specification Languages have been proposed for specifying reconfiguration rules. In Rosenmüller et al. (2011b) a DSL was proposed for specifying context rules, and what features should be bound/unbound in a given configuration. Events are not handled via context managers but proposed using monitoring code in the host language, in this case C++. Instead of handling changes directly on the configuration of the product, adaptations are handled by reasoning over features selected by the user, contained within a requirement containership. From the required features, the configuration is derived by then including features required according to feature relationships e.g. parent features, feature dependancies. One large drawback to this method is it unlike application features, context features are static within an application.

General purpose context-management systems have been used in DSPL to provide context. COSMOS as presented above has been used in the work of Parra et al. (2009). To use COSMOS, the authors proposed to model context-aware assets and map clauses to context nodes using context conditions against observables. System reconfiguration is handled by receiving COSMOS context notifications, which are linked to observables. If the cause is evaluated to be true, then the body of the clause including the change, the actually action to be taken; and the place, where the adaptation will take place, are realised.

Mostly these approaches only consider environments where single applications are running. When we consider modern mobile computing, its common to have multiple applications installed, and sometimes running on the device.

3.4.3 Summary

While context has been used in DSPLs before, we find several shortcomings of the previous approaches. Firstly, many of the approaches treat context as a different entity entirely of the system. This is usually carried out by using the context-aware managers as just an input into the DSPL for what features should be activated/deactivated. Because of this, many contexts are considered only in a static fashion, whereby if they are in the product, they are always active and being used. This causes two downfalls. Firstly, it makes the assumption that for every product, that context will be required. Just as discussed earlier, through stage configurations, not every feature may be included in a deployed DSPL application. This can make it harder to specialise the DSPL, as the context is not considered part of the DSPL directly, and therefore configured alongside it. Secondly, its possible that depending on context, it maybe useful to alter what contexts are being used. For example, if the battery on a device is getting low, it probably is not suggestible to keep sensing for the device's location using GPS.

A second issue is that, of the approaches that treat context as a feature, these do not exhibit the DSPL property of variation point changes during runtime as defined by Hallsteinsen et al. (2008). Because this, only foreseen contexts can be taken into account. Furthermore, Parra (2011) recognised the need for runtime context extensions. The authors discussed that in their approach context observables not defined in the initial aspect model cannot be added at runtime. Also, there is lack of runtime reuse of contexts across multiple products. This then can equate to several duplicate contexts running at once. It would be more useful if contexts can be added to the context manager that can also be used by other products.

3.5 Research Goals

From the literature review carried out above, we can conclude that SPL approaches have be used only for static GUI variability. We can also conclude that while there have been some proposed approaches to tackle both types of adaptations there are issues.

With the previous work in mind, and their shortcomings, we can therefore specify the goals of this thesis. In this thesis, our goal is to extend DSPLs to handle GUI variability, and also to treat context in the system as a standard feature.

3.5.1 GUI Variability Unification

In this thesis we investigate how variability in graphical user interfaces can be unified in DSPLs. In existing DSPL approaches there has been a focus on support runtime logic changes, normally within a single language or technology. While a GUI can be implemented within a single language, our research considers how this variability can be handled when the GUI is implemented using GUI documents.

Enabling the use of GUI documents within the DSPL process can offer advantages. Firstly, it should provide a unified representation of GUI adaptation. This unification has been an aim for previous work on program logic, and should also apply to GUI documents. By unifying adaptation, GUI adaptation can be applied either statically at compile-time or dynamically at runtime depending on the specific needs of that project.

Next, there are many considerations that need to be taken to in to account when considering GUI adaptation in DSPLs. One such consideration is configuration timing. Logic adaptations are applied when a DSPL reconfiguration occurs. With GUI changes, depending on the granularity of the changes, it may not be best for every refinement to be added at anytime. Within the lifecycle of a GUI, there are two main phases in which a refinement should be applied, on *inflation* or while it is *active*. The inflation of a GUI can be regarded as when the GUI is being constructed during the transition from one screen to another. In contrast, the active phase of a GUI is while the screen or GUI is currently active and visible to the user.

3.5.2 Context as a Feature

Additionally we claim in this thesis that context within a DSPL can, can be unified as a feature of the DSPL, both at the modelling stage, and how it is handled at runtime. Hinchey et al. (2012) recognise that on going DSPL research is attempting to enlarge variability modelling approaches to capture context descriptions and decision making. If context is considered a completely different element to the system, it requires different modelling techniques, and lacks the benefits that a DSPL can bring to the rest of the product. In previous work, DPSLs are driven by context by that the system reconfigures depending on what contexts are active. When considering context as a feature of the system, we not only consider context as state, but also the implementation that acquires infers is state. This means that each context should be capable of dynamic binding, just as other features in the system. By using dynamic binding with contexts, these components can then be used dynamically, allow the system to adapt its own context retrieval.

By considering context as feature we believe that benefits can be found. Firstly a benefit is that we can use a unified modelling language to represent context, the system in which it operates, and the changes it makes. This is beneficial as it allows it allows the developer to be able consider the relationships between the contexts on a conceptual level, but also implementation level. Another modelling benefit is be able to need only to understand one modelling syntax and semantics.

A second benefit, is that like other features of the system, these features should by able to be static and dynamic. In previous work, contexts are defined and deployed at compile time, making them unable to change at runtime. Particularly with mobile devices, if context is used to help improve an application, that should not just be confided to rest of the system, but also to it context retrieval and reasoning. By enabling the contexts to be dynamic and not just statically added to a product, we believe benefits to the system can be achieved.

3.6 Summary

In this chapter we have explored several existing approaches related to the work in this dissertation. Firstly we describe and discuss existing DSPL approaches. These approaches though do not concentrate any attention on GUI changes, and instead only consider program logic adaptation. Next, we considered how GUI variability has been handled in the past statically, and dynamically. While plastic UIs have been proposed to handle static and dynamic adaptation, these are strictly outside of an SPL approach. Lastly we considered how context is used in DSPLs and other context-aware applications. It is show that within DSPLs, context is normally handled as a static issue, which is restrictive.

With this, the state of the art of this thesis is concluded. In the following part, the contributions of this thesis are presented for both GUI variability and context handling.

Part II

Contribution

4 GUI Variability

Contents

4.1 Introduction

A central issue to adaptive GUIs and SPLs is variability. It is this variability that allows applications to be tailor made, and make dynamic changes to the application GUI at runtime. In this chapter, we wish to explore the different types of variability that can exist within the space of GUIs in mobile applications. As these GUIs can be implemented using a mixture of GUI documents and programming logic, we consider GUI variability that can be applied to both implementation approaches.

Also, because we are attempting to handle GUI variability both statically and dynamically, it is necessary to analyse the different issues that require attention. Together, this chapter aims to define the different requirements of our proposed approach in the next chapters.

This chapter is organised as follows: In Section 4.2, we describe different types of variability that can exist in the GUI of a mobile application. Then in Section 4.3, we discuss the different issues that need to be handled when dealing with both static and dynamic GUI adaptation. We then conclude the chapter in Section 4.4.

4.2 Types of Variability

Variability can affect the GUI in a number of different ways. GUI variability can occur to suit different localisation/internationalisation and design needs. Particularly in the instance of internationalisation and localisation, it has been proposed that products translated for new cultures become entirely new products (Nielsen, 1990; Russo and Boor, 1993).

While it is the case that this variability can adapt the GUI in an almost arbitrary way on an implementation level, this understanding can leave a very ambiguous view on the types of adaptation that can occur. We therefore attempt to analyse what types of variability can exist within the GUI for both static, and dynamic mobile applications. We do not attempt to consider all HCI variability types, including different frameworks/architecture, methods of input etc, but instead try to keep within the immediate area of the GUI including its look and feel.

Variability of GUI in SPLs has been previous investigated by Pleuss et al. (2012b). Authors analysed a university management web application by measuring the variability across the following concepts:

- **Presentation Units**. Variability of this concept was mapped to different screens e.g. HTML pages.
- **Dialogue of Presentation Units**. This concept considers the navigation of the web applications, considering the order of HTML pages.
- **UI Elements**. This considers the different HTML elements. For the variability, the authors extracted the elements from the HTML, and CSS documents.
- **Properties of UI Elements**. This concept represents the different logical properties of each element including default selection etc. This variability was extracted via attributes that are currently selected in a HTML page.
- **Dialogue of UI Elements**. In this concept, different input validations are included. These were extracted checking for a specific attribute in each HTML node.
- **Layout**. Spatial properties and structuring of HTML elements are part of this concept. In HTML files, the order of different elements and their size attributes were extracted.

- **Visual Appearance**. This includes the different visual properties of the HTML elements. For this concept, HTML element labels including div with id were collected.

The presented elements can form a solid basis for GUI variability of many different systems. We wish to take these different concepts and expand on them in the context of mobile applications. Furthermore, unlike many desktop web applications, mobile applications take advantage of different user gestures, and different screen sizes. This dimension of variability should be considered along with the other presentation related GUI variability.

4.2.1 Presentation Units

A presentation unit can be described as a collection of UI elements. Presentation units can differ in the amount of screen real estate they utilise, covering the complete screen, or alternatively occupy just a certain fragment of the screen. While when using GUI documents, it is possible to declare only single UI elements, more often it is the case that each document represents a presentation unit.

In the context of implementing GUIs using programming logic, we define a presentation unit as a single programming module e.g. a class in OOP used for implementing a view. These views similar to GUI documents can be used for implementing a single UI element, or a tree of UI elements.

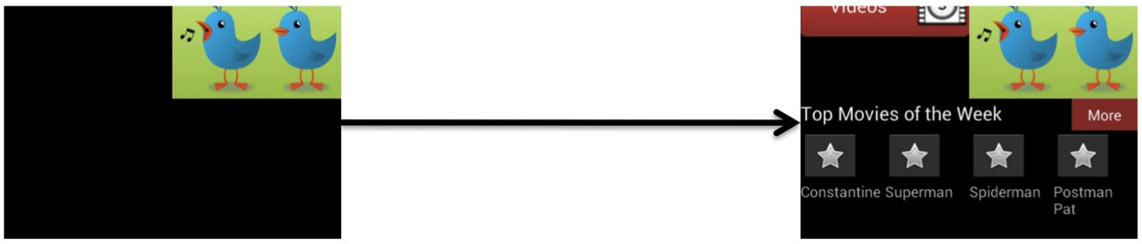


Figure 4.1: Presentation Unit Variability

If we take an example content store application, on the home page, content recommendations can be advertised to help indicate content they might be interested in. Since it is possible that not all types of content will be available, due to geographic position, recommendations for videos should only be be show if videos are available to buy and view.

4.2.2 UI Elements

A GUI is made up as a tree of different widgets, or UI elements. Widgets can take on either input, output, or both roles. Examples of these widgets can include buttons, editable textfields, labels, and checkboxes. Not all widgets have to be a visual element, but can also be used to help shape the GUI. We call these type of widgets, container widgets, as they are used for containing other widgets. Containment widgets can be used to shape the GUI by allowing widgets to be arranged in different ways including linear layouts, that is placing widgets either vertically or horizontally after each other, and relative layouts, that place widgets in relative positions in the container.

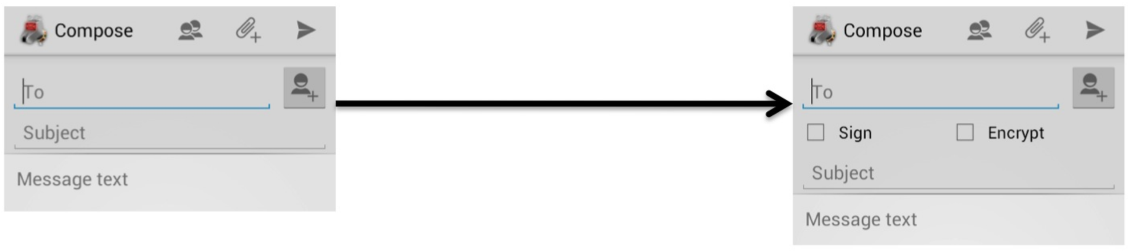


Figure 4.2: UI Element Variability

If we take for example an email application, the ability to sign and encrypt emails is not a feature many users need and use. Therefore, only in cases when the feature is needed, are the checkboxes included in the GUI.

4.2.3 Properties of UI Elements

A property of a UI element corresponds to different logical settings of that element. When considering certain UI elements like checkboxes, and radio buttons, there can be default selections, also textual hints in text fields to add the user.

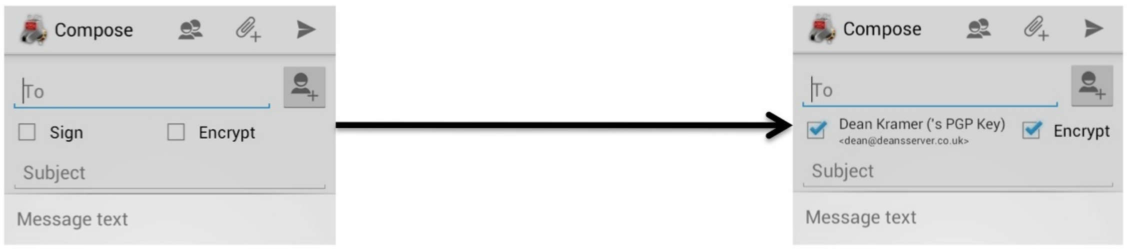
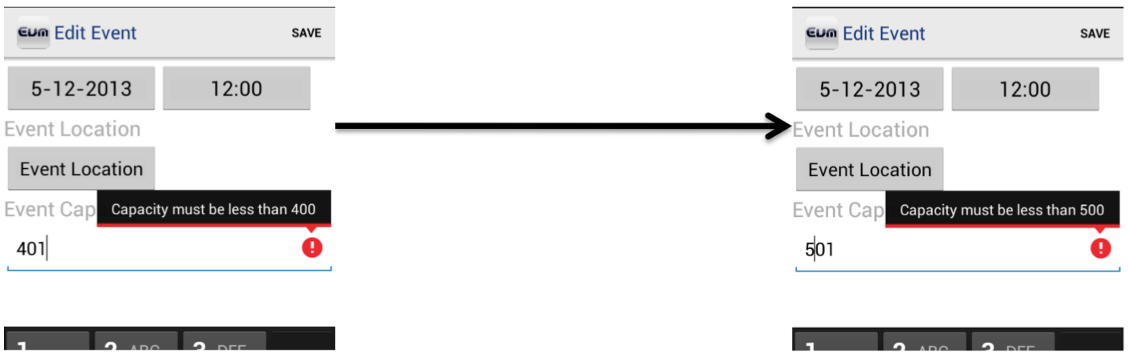


Figure 4.3: UI Element Property Variability

Taking the same screen as before, particular customers may want the email application to sign and encrypt by default. The application should still allow for non signed and encrypted emails when communicating with clients external to an organisation. This is then handled by having the signed and encrypted checkboxes checked by default.

4.2.4 Dialogue of UI Elements

By dialogue, we include reactions that require intervention by a user. This can be for input validation, or verification of data. Examples of these include maximum input lengths, and character types in text fields.



Capacity must be less than 400

Capacity must be less than 500

Figure 4.4: Dialogue of UI Elements Variability

In the example screen for adding and editing event information in an events management application, there may be a maximum limit to the amount of attendees that the event may have. When the user enters the capacity of the event, if the capacity is above the maximum an error can be shown requiring remedy.

4.2.5 Layout

The layout of a GUI primarily applies to the positioning and sizes of the different UI elements in the GUI. This also includes ordering of widgets, when contained in order dependant layouts. Layout variability can be used for a number of reasons. For instance it can be used for ensuring input fields are constant across different corporate applications. It can also be used for handling different localisation issues, for example dealing with left-to-right and right-to-left reading users (Russo and Boor, 1993). Lastly, due to the diverse sizes of mobile devices, and how users hold the devices can drive the user interface that enable easier interaction different aspects of the GUI.

4.2.6 Visual Appearance

For a UI element to be visible, visual properties are needed. Depending on the type of GUI element, different visual properties can be used to set the visual appearance. Examples of these include colours, text font type, borders, background images etc. Variability on this level can be used for enabling a level of visual consistency with other software systems or themes. These different systems or themes can include having a consistent look across multiple corporate applications (Brummermann et al., 2011), or with a given platform.

4.2.7 Orientation

Mobile applications typically run on a range of different screen sizes including phones, and tablets. Depending on the horizontal and vertical requirements of the GUI, different orientations can be used. These orientations can often either be set, that is, permanently in portrait or landscape, or can also allow for both orientations. In the cases where both orientations are enabled, it might be necessary to also alter the orientations of different UI elements, or alter their position to enable easier handling.

4.2.8 Behaviour & Interaction

Behaviour and interaction is an important aspect of a mobile GUI. GUIs carry out different tasks for the user, including intercepting gestures from the user, and completing actions when particular events occur. Just like layout, interactions can be related to localisation issues. For instance, if a slide out menu is to be displayed on the left of the screen, for left-to-right reading users, it makes sense to intercept sliding actions from the left edge of the screen to the right. Alternatively, if a menu is to be displayed to the right of the screen, for right-to-left reading users, it then makes sense to instead intercept a sliding action from the right edge to the left of the screen.

As an example if we consider the email client application, different gestures can be used to signal different quick actions to carry on to each email e.g. deletion, or move to junk. In this application then, depending on the direction of swipe, left or right, a different action can take place.

4.2.9 Compound Variability

These variability types can be used in isolation, or also compounded and used together. In some cases, to ensure a consistency across a GUI, variability in the GUI can encompass more than a single dimension of variability. For instance, when handling variability to enable a consistent look and feel for corporate applications, it is reasonable to assume that not only the visual appearance of UI elements will be adapted, but possibly the layout also.

4.3 Static and Dynamic Adaptation

GUI variability in DSPLs, as proposed in this thesis, should be capable of being applied both statically at compile time, and dynamically at runtime. Runtime GUI adaptation can be triggered by a combination of automatic e.g. context changes, or manual changes e.g. switching by the user. When considering adaptation at runtime, there are a number of different issues that need to be addressed to help ensure the GUI remains usable by the user.

4.3.1 Timing

During the reconfiguration of a DSPL, the running application is adapted instantly. While logic adaptations can be applied instantly, this might not be desirable for all GUI adaptation. Particularly when considering the usability of the GUI, it can become increasingly confusing to the user when UI elements appear and disappear on the screen (Holzinger et al., 2012). It could still be the case that updating the properties of already existent UI elements on the GUI may not cause this issue. Therefore, we propose that the developer should be capable of applying GUI adaptation at different times to suit different adaptations.

To support different GUI adaptation times, we consider two different phases where adaptation in a dynamic GUI may be applicable:

1. **On Inflation.** The inflation of a GUI can be regarded as when the GUI is created, either when the application first starts, or during a GUI transition from one screen to another. It is possible that large amounts of adaptation e.g. presentation units, may be best suited for this time in the GUI lifecycle.

2. **While Active.** We consider that a GUI is active when it is currently visible to the user in the foreground of the device. During this stage in the GUI, smaller adaptation e.g UI element properties could be more suitable e.g. deactivating a download button in the situations where an internet connection is not present.

4.3.2 Adaptation Isolation

A GUI can require different amounts of adaptation ranging from a UI element property alteration to whole presentation units. These adaptations should be carried out only on the GUI widgets requiring adaptation. Widgets in the GUI can take many different roles, and can carry out tasks for example video playback. If during an adaptation video playback is stopped, or affected by pausing, this could potentially be frustrating to the user. Therefore, unless a widget is being adapted, it should be left to function normally, even during adaptation. This should lead to less noticeable adaptation transitions. While for many adaptive approaches, this isolation can be straight forward to handle, for GUI documents this is less trivial. Existing GUI frameworks commonly parse each GUI document at runtime when it is required by the system. These are parsed whole and create the GUI from all the properties that exist in that document. There is often no ability to parse fractions of these documents, applying only what properties need to be changed. We therefore need to ensure that despite reading variants of the GUI as a whole document, we still only adapt the UI elements that require adaptation.

4.3.3 GUI State

GUIs in a mobile application are rarely stateless. State in the GUI can be altered directly by user input for example text in a editable textfield, or can be altered indirectly for example the position of a video playing in a video container. When an adaptation occurs, we need to ensure that state of the different UI element remains. State in a GUI during adaptation can be retained in a number of ways. The first method is that adaptations of UI elements only update the properties of an instance, leaving the rest of the properties the same. The second is to store the state of the widget, carry out the adaptation, and copy the state back. In the context of using GUI documents, the second option will have to be used. Having said this, it may not always be possible to store all UI state data, due to different platform constraints. If this the case, the developer will have to decide if the variability will have to remain static.

4.4 Conclusions

Concluding this chapter we consider the different design decisions that have to be taken due to the different aspects discussed earlier. Firstly, we can see that there are many different types of variability that should be supported in the proposed approach. As discussed, consideration is needed for adaptation to be applicable at different times. This should be part of the project configuration, thus allowing the issue of timing to be customisable on a per product. Timing can be applied either to the GUI document itself, or to the GUI controller that uses the GUI document. To ensure GUI state is retained, we need to enable the developer to easily handle state transfer between old and new widget variants. This can be handled by allowing the developer to declare and reuse fragments of logic for transfering state between any arbitary widget types.

4.5 Summary

In this chapter, different variability types in mobile GUIs were introduced and described. Next, different issues relating to the application of the different variability types statically and dynamically were presented. Finally we consider some of the design decisions required for the proposed approach based on the different adaptation issues specified. The approach for defining and implementing GUI variability is described in the following chapter.

5 Design Phase

Contents

5.1 Introduction

DSPLs have provided a method for applying SPL techniques for the development of adaptive software, which can allow for a unified development of adaptation. There are still a number of challenges that exist with DSPLs. Firstly, the types of adaptation focussed by DSPLs has concentrated around program logic. A unified approach for the GUI however has been neglected. Secondly, context has largely been treated as a different entity to the system. Consequently, context has required separate modelling, and is often handled differently to the rest of the system at runtime. Dealing with these challenges is a primary focus on our DSPL development approach, which is presented in this chapter.

We present our approach following the *Domain Engineering*, and *Application Engineering* processes used in typical SPLs, as depicted in Figure 5.1. Domain engineering refers to the design and development of the product commonality and variability. Application engineering on the other hand refers to the creation of software products, reusing artefacts defined and implemented in the domain engineering process.

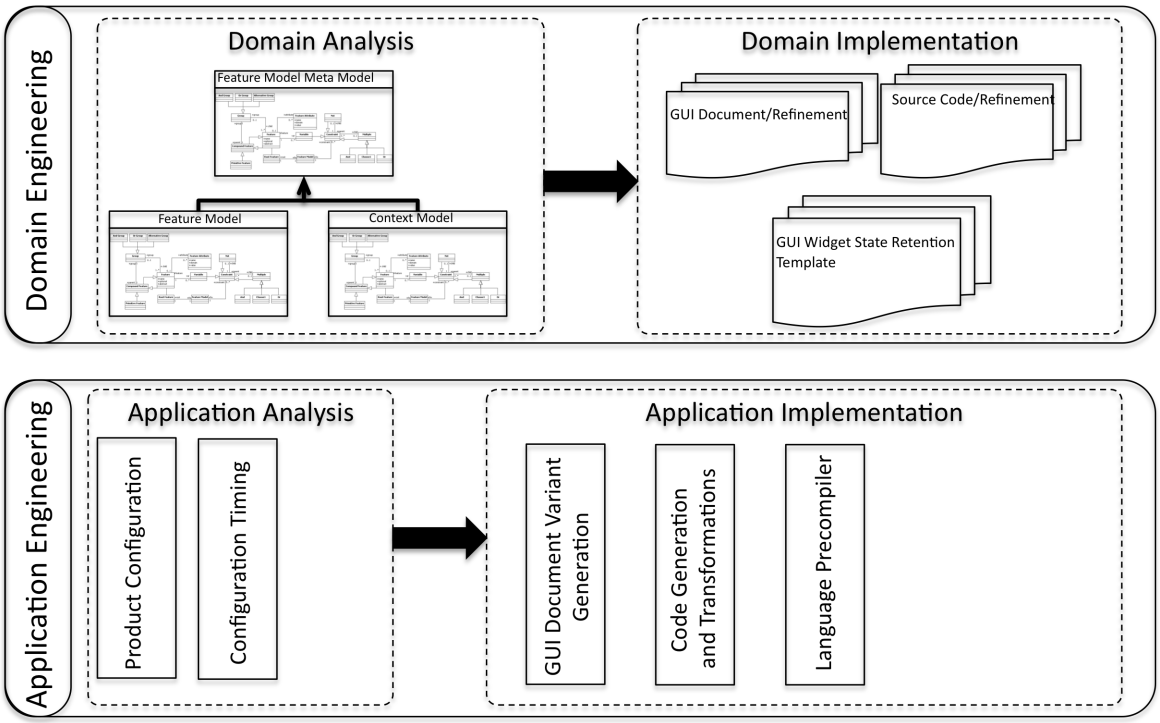


Figure 5.1: Overview of the DSPL Approach

Within the domain engineering stage, there are two distinct phases, *domain analysis*, and *domain implementation*. For the domain analysis, we introduce an approach for modelling system variability and context using extended feature models. Domain implementation is made up of many elements including a Feature-Oriented approach to handling variability in GUI documents, and GUI related source code variability.

Next, in the application engineering stage, we have the *application analysis*, and *application implementation* phases. During the application analysis phase, we configure the product using static and dynamic composition. Included is the ability to control when different GUI adaptations can be applied. Application implementation encompasses the derivation process that produces a product based on the assets implemented in the domain engineering phase.

This chapter is broken down in the following sections: Section 5.2 presents the domain engineering process of the DSPL, introducing the method of variability modelling, and implementation. Section 5.3 then describes how an application is derived from the SPL, including code transformation and generation. Finally, the chapter is summarised in Section 5.4.

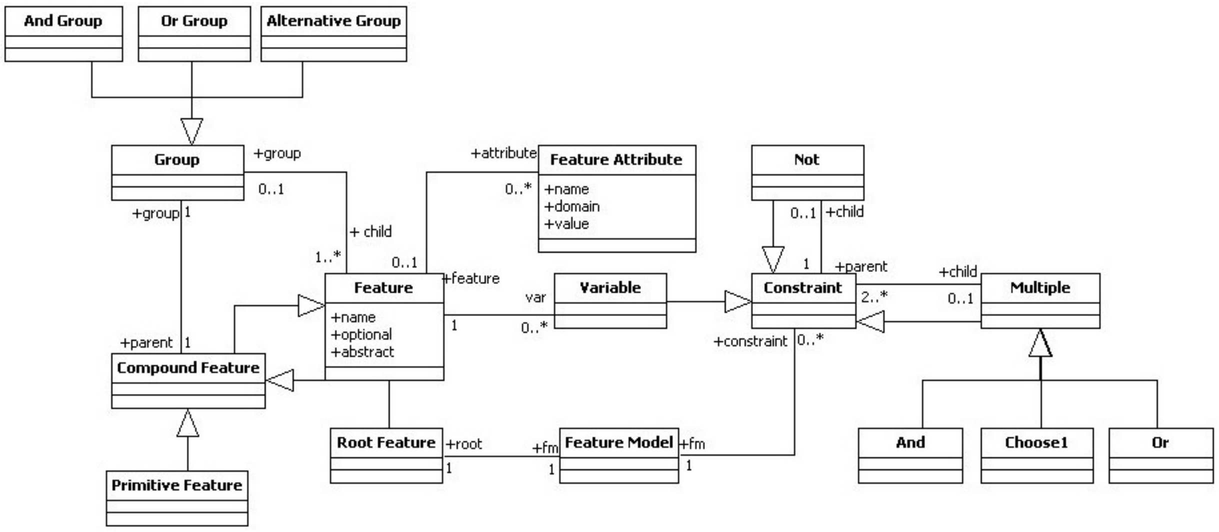


Figure 5.2: Feature Model Meta Model

5.2 Domain Engineering

In this section, we describe the domain engineering process of our approach. Within this process, we focus on domain analysis, and implementation. For the domain analysis, we focus on the need to model the variability of the DSPL, and the different contexts that the system needs to recognise at runtime. These models are modelled using extended features, which allow us to model the application and context without the need for supplementary models.

Following the domain analysis, we can implement the commonality and variability. In our approach, we apply Feature-Oriented Programming concepts to enable the ability to implement GUI document variability. We handle this variability using document refinements.

5.2.1 Feature Models

Feature modelling has become the de facto modelling language for SPLs, whereby system commonality and variability is modelled in terms of features. One issue with using plain feature models is that while the domain engineer can express feature relationships, it does not allow for more information about features to be expressed. Because of this, our work is based around the use of *extended feature models*. Extended feature models allow for extra information to be attached to features (Benavides et al., 2010). Extended feature models add the concept of *feature attributes*, which are attached to features. Feature attributes commonly are formed of a *name*, a *domain*, and a *value* (Benavides et al., 2005).

In Figure 5.2, we present a meta model of our feature model incorporating feature attributes, which is an extension of the GUIDSL feature model, used in FeatureIDE (Kastner et al., 2009). Our extension has been added onto the meta model presented in (Thüm, 2008). In the meta model, all compound (including the root feature) and primitive features contain a unique name, whether the feature is optional or not, and if the feature is abstract. Abstract features are features that contain no implementation artefacts, and have been proposed as being used purely to help structure the model and make it more readable (Thum et al., 2011). Compound features are parent features, that contain a number of subfeatures, using a group relationship of Or, And, or Alternative. Each feature can contain any number of feature attributes, each containing a name, domain, and value, following previous proposals (Benavides et al., 2010).

The feature model also can contain any number of constraints. Constraints are built recursively, allowing for constraints to be formed of other existing constraints. Connectives such as Not, And, Or, and Choose1 can be used. The Choose1 connective is used when only a single constraint is true while the rest are false.

Using this meta model, we can model the main application. Following the modelling of application, we need to express the context model for the running DSPL application.

5.2.2 Context Model

The context model is used to help represent the different conditions that can cause a change in the application. By using a context model, different raw context data can be abstracted to form high level situations. When modelling context, Bettini et al. (2010) identified considerations that should be considered, some of which are the following:

- **Heterogeneity and mobility**. Contexts are not all the same, and can differ in a number of ways. These different ways include the rate in which they update, the type of data they obtain, and the method in which they obtain the context data. Context acquisition should be adaptable to a changing environment, especially when in mobile applications.

- **Relationships and dependencies**. Different relationships can exist between contexts. Particularly when dealing with higher level context data, changes in lower level context information can call a value change in dependant higher level contexts.

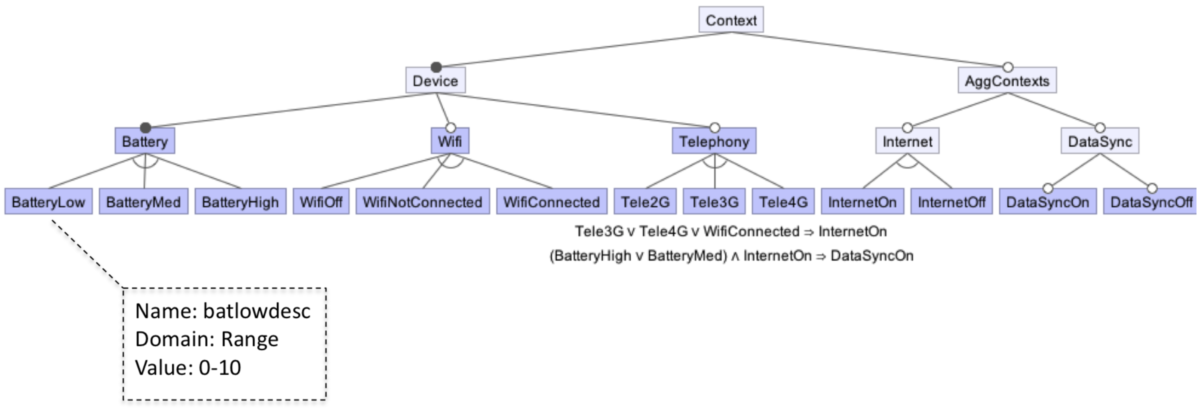


Figure 5.3: An Example Context Model

- **Reasoning**. Different higher level context information should be able to be obtained by lower level atomic contexts. This can be broken down into two parts. Firstly, context information should be able to be derived by raw context data, by binding context definitions to specific context data values. Secondly, higher level contexts should be able to reasoned over, by the combination of other contexts, producing compound contexts.

- **Usability of modelling formalisms**. Context models are created by developers, and then used in context management systems. Because of this, context models should provide an easy method of modelling real world concepts to constructs in the model.

Based on these requirements we propose to model context using the feature model meta model presented in Figure 5.2. In Figure 5.3, an example context model has been depicted. This model captures the different states that can exist for 5 different context types, *Battery, WIFI, Telephony, Internet* and *DataSync*. These contexts can then have each of their values defined in terms of the raw data necessary for such a context value.

To specify different elements of the model, three categories of features are used in the context model including:

- **Group**. Group features are those designed to aid in readability of the model. This means that they do not relate to any specific context, but can be used for grouping appropriate contexts together in specific categories. These feature types are represented as abstract features, and therefore are shown in the feature diagram as a lighter colour. As an example in the context model, the five contexts that are visible are regarding the context of the device. Therefore we can put all these contexts within a group called *device*.

- **Context**. Context features correspond to atomic contexts. For example in Figure 5.3 you can find there is a context for *Battery, Wifi,* and *Telephony*. These features can either point to concrete context implementations used later for sensing that specific context, or they may be empty, using an already present implementation in the middleware. By using the feature relationships including mandatory, optional, Or, and Alternative, the developer can control what contexts must always be active, or not. Taking the context model as an example, the battery context has a mandatory relationship, so it should be always be active. The other contexts though can be switched on or off because of their optional relationships.

- **ContextValue**. These feature types are found as the terminal nodes of the feature model. A context value is a particular high level value that can be deduced from the context engine. These values can be seen as abstracted values based on raw context data, or other context values. These abstracted values can aid in defining understandable context rules and aggregations, and allow the context acquisition engine only to report when those values are found. In our model, contexts can only have a single state at any given time, and this is expressed using the alternative feature relationship with the context feature. Finally, to express the actual raw context data that is abstracted by the particular context value, we use *feature attributes*. In the feature attribute we define the type of raw data that needs to be handled, including a name, the type of data, and the raw data value(s) to check against. As an example, we have a ContextValue feature called BatteryLow representing when there is little charge remaining in the battery. This ContextValue has a feature attribute describing what raw data represents a battery with low charge, which is a range of zero to 10 percent.

In a context model, the hierarchical structure of the model is important in expressing different constructs of the model. When modelling atomic contexts, there are a set of constraints that need to be observed. Firstly, ContextValue features must have at least one feature attribute. Secondly, Context features are always direct parents of ContextValue features. Lastly Group features are either parents of Context features, or other Group features.

Context Aggregation

Contexts discussed so far are those that we describe as atomic contexts. Atomic contexts are those that deal with a single source of context information that can function in isolation. There are times however when a context is not based on a single piece of data or situation. We call these aggregated contexts, *composite contexts*. A composite context is one that is based on the conditions of a collection of other contexts. A composite context can be made up of a collection of atomic, or other composite contexts, to form higher level knowledge of a given situation.

As an example of context aggregation, let us consider the composite contexts in the context model in Figure 5.3. In this model we have two composite contexts called *Internet* and *DataSync*. The Internet context allows us to monitor if the device currently has an internet connection over either the Wifi, or telephony connection. The DataSync context allows the device to know what situation is best to synchronise data for an application, based on the internet connection and battery life.

To model composite contexts in the context model, these are defined in a similar fashion to atomic features, but with some differences. Composite contexts can be defined using Context features, with ContextValue features being used to specify each state of the context. How the definition of composite contexts differs is that feature attributes are not required, as they are with atomic contexts. After the composite context has been added to the feature model tree, aggregation rules are needed. These rules are expressed using propositional constraints.

If we take the current composite example, first we can express the composite context internet connection context as:

$$Tele3G \vee Tele4G \vee WifiConnected \Rightarrow InternetOn$$

Next, to write the aggregation rule for the DataSync composite context, we can express this as:

$$(BatteryHigh \vee BatteryMedium) \wedge InternetOn \Rightarrow DataSyncOn$$

Once the context model is complete, the context events can be modelled.

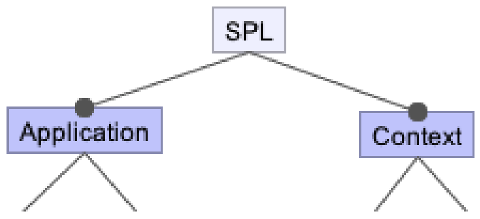


Figure 5.4: Feature Model Aggregation

5.2.3 Context Event Rules

DSPLs allow for dynamic reconfiguration, which can be driven by context changes. To enable this reconfiguration, it is necessary to model these context events that cause a configuration change. Reconfiguration of the DSPL can include the activation/de-activation of application features, and contexts. By changing the activation status of application features, the application can be adapted to behave or be visually different. Changing the status of Context features on the other hand can define if a context is needed or not by the system.

Context events are expressed much in the same way as context aggregation rules, using propositional constraints. These rules are used to bridge features in the application feature model, with features in the context feature model, acting as intra-model constraints (Acher et al., 2009). By using this method, we can control what application features are activated when a feature is activated in the context model, e.g. a ContextValue feature.

To model these intra-model constraints, we create an aggregate feature model. This aggregated feature model is formed of the application feature model, and the context feature model. Each model becomes a mandatory child of a larger model as depicted in Figure 5.4, proposed by Acher (2011). This aggregation is carried out automatically. The developer can then define these intra-model constraints on the aggregated feature model. Taking our scenario application into consideration, videos and music should only be able to be downloaded providing there is enough storage space on the device. If there is not enough space, the content is streamed instead. This then requires a context for monitoring the amount of storage on the device, which can be used to adapt the application to only allow content streaming. We could express this rule as:

$$StorageLow \Rightarrow Streaming$$

Also, as described earlier, context events can be used for defining whether a context is required by the application or not. As an example, in an application that is running low on battery power, it could be beneficial to stop monitoring the current location of the device. This can be express as:

$$BatteryLow \Rightarrow \neg GPS$$

This modelling approach shares many of the same ideas proposed by (Moisan et al., 2012). Where we differ is that we use extended feature models instead of basic feature models. By using extended features models, we can attach context values to the contexts using feature attributes. If basic feature models are used however, these context values need to be defined elsewhere in a separate model or definition. Also, other works have shown, reasoning on feature models including feature attributes increases computation complexity (Cordy et al., 2013). Instead of reasoning using the feature attributes, our approach uses feature attributes to just store the context values required for the context acquisition system. Reasoning is therefore only carried out over the features, making the model easier to reason over.

5.2.4 GUI Document Variability

Following domain modelling, domain realisation can take place. We start this process by implementing the GUI variability. As our focus is on use of GUI documents in GUI development, we look at how we can express this variability in this type of artefact. In our approach, variability is handled using positive variability, whereby different variable elements are added to a common base based on what features are present (Voelter and Groher, 2007). Variable elements of GUI documents are expressed using GUI document refinements, and are applied in a stepwise fashion.

Document Refinements

To implement variability in GUI documents, we propose to follow the approach used in FOP. Refinements to GUI documents bring about changes to the GUI they represent. Within a refinement, widgets displayed to the user can either be added, or can be altered to have a different property e.g. colour, shape, or size. Document refinements are implemented using physical separation of concerns, whereby each refinement is held within its own file. These refinements are contained in feature modules, along with other source code refinements, and documentation for feature cohesion.

Within each GUI document, nodes in the GUI tree are expected to have a unique identifier. These identifiers can then be used for searching and identifying nodes within the tree. This is needed when specific ordering is needed, as described later. In Listing 5.1, we depict a GUI refinement that refines the home page of a content store application to add GUI elements to allow the user to browse video content types. As shown, we see that for Android applications, GUI element identifiers are defined with the `android:id` node attribute. When adding additional widgets to the GUI tree, all parent nodes are needed. In the example we can see that a button named with `@+id/videos` has been added to the tree. To add or override a node attribute, the attribute and its value is needed in the refinement node. If that attribute is then found to exist during composition, it value is overridden, otherwise it is added as an additional attribute. Within a refinement, the only attribute that can not be overridden are node identifiers.

Listing 5.1: An Android GUI Document refinement

```
<FrameLayout
       android:id="@+id/mainFrame">
       <LinearLayout
              android:id="@+id/mainlayout">
             <LinearLayout
                    android:id="@+id/contenttypes">
                    <Button
                         android:id="@+id/videos"
                         android:layout_width="160dp"
                         android:text="@string/videos" />
             </LinearLayout>
       </LinearLayout>
<LinearLayout
       android:id="@+id/adverts">
       <LinearLayout
            android:id=@+id/videoAd
            ....>
             <TextView
                 android:id="@+id/TopMovies"
                 android:text="@string/TopMovies"
                 ..../>
       </LinearLayout>
</LinearLayout>
</LinearLayout>
</FrameLayout>
```

Refinement Ordering

Ordering can be an important issue when handling GUI document refinements. Generally in GUI documents, the relative position of GUI elements is defined by the document structure. Therefore, if a button in a vertical part of the GUI is defined before a text field in a GUI document, it will also be placed this way on the GUI. Ordering can partially be accomplished by composing features in a particular order, but this does not help when a UI element needs to be placed before an element in the base feature.

Tools for composing non source code assets, for example XAK (Anfurrutia, Díaz and Trujillo, 2007), partially reduced this problem with the ability to place items either at the beginning (prepend) or at the end (append) of the parent node. This is sometimes not enough, and the issue of ordering can still be difficult to achieve with shallow trees. In order to alleviate this issue, we propose a set of keywords that be used for node hooking. We propose the ability to insert nodes either before, or after a given node within a shared parent node. This can be achieved on multiple nodes, by placing the list of nodes within that particularly keyword block.

Because many software development kits (SDKs) allow for graphical editing of GUI documents, it is good to avoid parsing errors caused by using tags/keywords that may try to be read by that SDK. To do this, our keywords are used within source code comments, which should allow for the GUI documents to be read by SDKs without parsing errors. In Listing 5.2, we depict two examples of placing a button both before, and after a particular widget in an Android GUI document. In our scenario application, we would use the `after` mode for both Music and Video buttons to make sure each button is underneath the Applications menu button. Each set of widgets that need to be placed before or after a widget need to be placed within a `@start` and an `@end` comment. In each comment, the type of position needs to be specified, `before` or `after`, followed by the identifier of the widget. If no widget identifier is specified, or widget specified does not exist in the same parent node on both trees, the keywords serve the same purpose as XAK, whereby the set of widgets will be placed either at the beginning or end of the parent nodes set.

Using these keywords and a relative composition order, widgets can be more precisely placed within a GUI document. The relative composition order is the order in which features should be composed together. This order does not just affect the composition of GUI documents, but all artefacts in the feature module, including source code.

Listing 5.2: Refinement Ordering Tags

```
<!-- @start before android:id="@+id/btnApps" -->
 <Button
    android:id="@+id/btnVideos"
    android:layout_width="match_parent"
    android:layout_height="59dp"
    android:contentDescription="@string/videos"
    android:text="@string/videos" />
    <!-- @end before android:id="@+id/btnApps" -->

 <!-- @start after android:id="@+id/btnApps" -->
    <Button
    android:id="@+id/btnVideos"
    android:layout_width="match_parent"
    android:layout_height="59dp"
    android:contentDescription="@string/videos"
    android:text="@string/videos" />
    <!-- @end after android:id="@+id/btnApps" -->
```

5.2.5 Source Code Variability

While static visual properties of the GUI are important, other elements of the GUI, handled in source code, may also exhibit variability that needs implementing. Variability can include adaptation to the behaviour of the GUI or any arbitrary part of the application. This source can refer to the controller and/or the model, in the MVC. Each controller handles the different events from the GUI, and can update the model or alter the view in the application. Here we discuss how the developer can handle GUI related variability implemented in the main source code.

In this dissertation, we do not propose a new approach for modularising source code. Instead we use existing language approaches, like Feature-Oriented Programming (FOP). By using this language based approach, the developer can realise source variability using refinements, allowing for a single method for realising product variability. How refinements are handled though are not always suitable for GUI adaptation. Method refinements for example are only executed when the refined method is invoked. These methods may or may not be executed, but GUI adaptation often needs to be executed straight away. Therefore, we need an approach to modularise GUI adaptation that will require execution when the DSPL reconfigures.

We support this GUI adaptation by structuring our GUI controller class to use two specific methods. The first method is to handle all GUI initialisation operations relating to the GUI document, with the second dealing with other arbitrary GUI adaptation.

GUI Document Initialisation

GUIs are created at runtime by parsing a GUI document into a tree of GUI widgets, a process that has been called *inflation* (Kramer et al., 2013). These GUI documents can either contain the entire GUI, or can contain a reusable fragment that maybe used several times within the same GUI. An example of a reusable fragment includes the GUI layout for a single row in a scrollable list.

When displaying a GUI tree or subtree, inflating the GUI document may not be the only operation needed. There can be a number of initialisation operations that need to be executed. Examples of these operations include adding onClickListeners to buttons for handling user touch events, setting class fields to widget object instances etc. These operations are only typically required when a new GUI screen is created, normally when a particular controller class is instantiated. With dynamic adaptation however, some initialisation operations may need to be carried out during adaptation, due to GUI tree changing.

Executing the same set of operations on every adaptation is not a good option. A single screen can be built using multiple GUI documents, therefore there can be multiple portions of the screen adapting independently. It is also possible that within a GUI, there are several instances of the same GUI document in a single screen. It would be better if only the operations required for an adapted portion of the GUI are executed. This can be handled by separating these operations into separate class methods, one for each GUI document. These methods are then only invoked if that GUI document has been inflated during a screen transition or adaptation.

Listing 5.3: Initialisation Method Refinement

```
public void onCreate_homescreen(ViewGroup vg) {
original();
Button btnVideo = (Button)vg.findViewById(R.id.btnVideo);
    btnVideo.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        gotoVideoStoreScreen();
      }
    });
}
```

Each of these methods should be named `onCreate_` followed by the name of the GUI document it initialises, with the GUI tree passed as an input. In Android the GUI tree passed is a `ViewGroup` object. As different widgets are added to the GUI document in different features, refinements to this method can be defined to initialise those widgets added. In Listing 5.3, we depict a refinement of the initialisation method for the GUI document `homescreen.xml` for our scenario application. In this method, we are adding a onClickListener to the button for Videos, named btnVideos. As you can see, because this method is not a base method, but a refining method, we use keyword `original` as used in FeatureHouse to specify when the statements should be executed, either after the last refinement or before. This ensures that other intialisations in the refinement chain are invoked first.

These initialisation methods are not just used for dynamic adaptation, but can be reused within the class. The developer can then use these methods for initialising the GUI when it is created in the first place.

Other GUI Adaptations

GUI adaptations can encompass a number of different types of changes. Not all adaptations to the GUI may be visual, or implemented using GUI documents. Examples described earlier in this thesis include gestures that may be used within an application. These operations may need to be carried out both when a GUI is created, or later on during a reconfiguration.

For these operations, we propose these be placed within a method named `onGUIConfiguration`. This method can be refined in class refinements to modularise these operations according to specific features. We can also call the method within the class constructor, if the operations are needed when the GUI is created. We depict an example in Listing 5.4. In this example, we are adding a gesture that removes a given element from a list. This gesture is activated when the user swipes a list element from one side of the screen to the other. The operations within a `onGUIConfiguration` method are not automatically reversed when a given feature becomes inactive. To avoid issues where changes become irreversible, statements to set that property to its default value should be defined as the base version of this method.

Listing 5.4: An example onGUIConfiguration Method

```
public void onGUIConfiguration() {
  SwipeDismissListViewTouchListener touchListener =
          new SwipeDismissListViewTouchListener(
                  listView,
                  new SwipeDismissListViewTouchListener.OnDismissCallback() {
                     public void onDismiss(ListView listView, int[] reverseSortedPositions) {
                         for (int position : reverseSortedPositions) {
                             adapter.remove(adapter.getItem(position));
                          }
```

```
                        adapter.notifyDataSetChanged();
                    }
                });
listView.setOnTouchListener(touchListener);
listView.setOnScrollListener(touchListener.makeScrollListener());
}
```

5.2.6 GUI State Retention Templates

It is important that the state of the GUI is retained during an adaptation. GUI state can encompass many different values for example text contained in an editable text field, or whether a checkbox is checked or not. Because updated widgets are effectively swapped from the current to the new widget variants, it is therefore necessary to move state from the old to the now current widgets.

To handle state transfer, we use code templates for each widget which can be used for handling state transfer between old and new widget variants. A template sample for TextField widgets is depicted in Listing 5.5, with the old and new widget variant references being contained within "<>" brackets. Each template is contained within it own file, with the filename being the name of the widget class it handles. During product derivation, templates are then used on the basis of what widget types are adapted at runtime. If a widget type is not being adapted, that template is therefore not used.

Listing 5.5: Code Template for some Android Widgets

```
//Template for Label and TextField Widgets
<NEW_WIDGET>.setText(<OLD_WIDGET>.getText());

//Template for ImageView Widgets
<NEW_WIDGET>.setBackground(<OLD_WIDGET>.getBackground());

//Template for Checkbox Widgets
<NEW_WIDGET>.setChecked(<OLD_WIDGET>.isChecked());
```

Because each of these templates are created for each widget, they can be reused easily over multiple SPLs. These templates for many developers then may only need to be defined once, and are then reused many times over different DSPL projects.

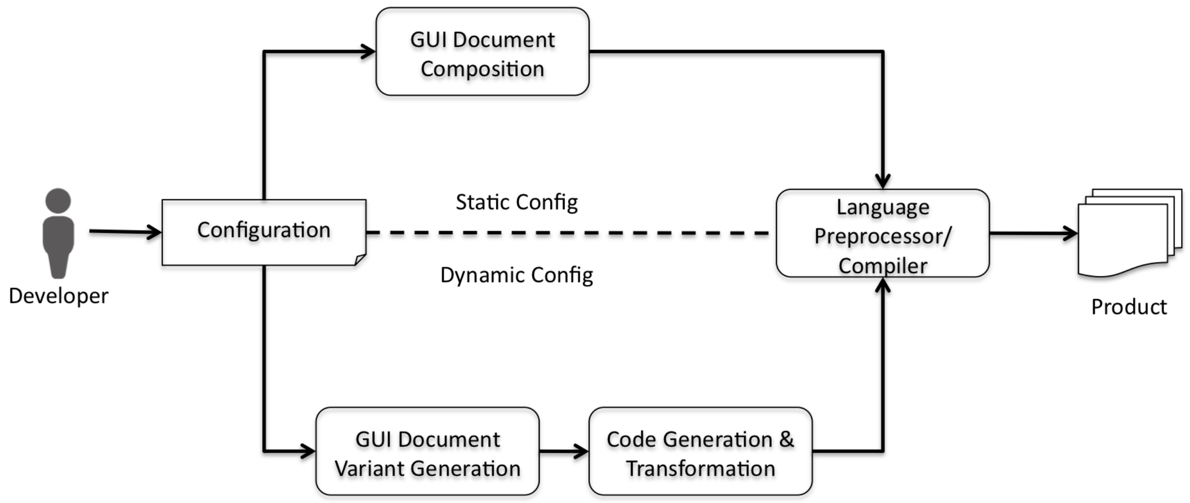

GUI Document Composition
Developer
Configuration
Static Config
Dynamic Config
Language Preprocessor/ Compiler
Product
GUI Document Variant Generation
Code Generation & Transformation

Figure 5.5: Product Derivation Phases

5.3 Application Engineering

In this section we describe our application engineering process, used to derive software products. This process is illustrated in more detail in Figure5.5. First the product configuration needs to be defined. Following the configuration, the process splits based on if the product is dynamic or static. If the product is a static configuration, only the GUI documents require static composition. A dynamic product however requires the different GUI document variants to be generated. These variants are then used to handle the runtime variability of the GUI documents. Following this step, code generation and transformations are required to enable runtime adaptation of the GUI. Finally, for both types of product, static, and dynamic, the source code will need to be preprocessed using the appropriate language preprocessor.

5.3.1 Configuration

The first step in product derivation is configuration. When creating a configuration for a DSPL, the developer needs to select what features are present in the final DSPL application. These features can either be static, whereby they are bound at all times in the runtime application, or dynamic, whereby they are included with the application, and bound only when needed. This means that some features which should never be bound in a running instance of the DSPL can be not included in the application. Features that are then not included in the application are no longer needed.

Compound Features

In a SPL, a product can be made of many features. While DSPLs give the ability to bind features at runtime, in many cases, not all features require dynamic binding. Dynamic binding can be used for every feature, however this has been shown to have a penalty in terms of compositional overhead (Rosenmuller, 2011). To help minimise compositional overhead while keeping necessary runtime flexibility, Rosenmuller (2011) proposed combining static and dynamic composition, using *dynamic binding units*. Dynamic binding units (DBU) are essentially compound features that are created by statically composing features together. These compound features are then used at runtime for dynamically binding feature refinements. By using DBUs, the runtime variability of the system is reduced, requiring the aggregated feature model to be refactored.

We refactor the aggregated feature model using the steps proposed by Rosenmüller et al. (2011a):

1. Remove all dead features that can not be selected in the runtime configuration. Dead features include others features in an alternative variant point. These features should then also be removed from existing constraints with other features in the model.

2. Compound features are added to the feature model. This is accomplished by creating a feature for every dynamic binding unit declared. These features then replace the containing feature closest to the root feature. Next, that replaced feature is added as a mandatory child of the compound feature. In the instances where the root feature is included in a compound feature, the compound feature is then added as a mandatory child of the root feature.

3. All features contained within a dynamic binding unit including their subtrees are moved to become mandatory children of the appropriate compound feature. This causes all features to be selected when a compound feature is selected.

4. Lastly, constraints that are no longer needed are removed.

Following the feature model refactoring, it is possible to reduce the feature model by removing mandatory features. If mandatory features are removed from a compound feature, the constraints of the feature model will need to be updated by replacing the removed features with their compound feature.

As we described earlier, features that are not a configuration are not present in the final DSPL system. This is based on the existence of each feature in a DBU. All features that are not in a dynamic binding unit can therefore be considered not part of the product.

Adaptation Timing

Adaptation Timing in an important issue as discussed in Chapter 4. We proposed that GUI adaptation should be applied during two different phases in the lifecycle of a GUI, *on inflation,* and while active.

By default, all dynamic GUI variability is handled at inflation. To enable adaptation to an active GUI for a DBU, explicit declaration is required. We declare every GUI controller class that can be adapted while currently active. This declaration applies to all types of GUI adaptation, including operations placed in the `onGUIConfiguration` method.

Static Products

A configuration can lead to both a static and dynamic product. As the product configuration is based on DBUs, the product can be understood to be static if only a single DBU has been declared. If the configuration is for a static product, only static composition is required. Also, as there is no dynamic variability in the system, context-awareness and runtime reconfiguration is not needed. With this in mind, runtime DSPL middleware, context model, and context rules are not needed for this running application.

The first step required is GUI document composition, as shown in Figure 5.5. This example is taken from our scenario application showing the menu adaptation for different content. This step takes the different refinements in composition order, and superimposes each refinement in a stepwise manner, as explained later. Following this step, the product is left with single variants of the different GUI documents in the SPL. After all GUI documents have been composed, composition of the main application source code needs to take place.

Composition of the main application source code is handled by the appropriate language preprocessor/compiler. These preprocessors include the FeatureC++ compiler (Apel et al., 2005) for C++ code, rbFeatures preprocessor (Günther and Sunkle, 2012) for Ruby code, and JAK (Batory et al., 2004) and FeatureHouse (Apel, Kastner and

Lengauer, 2009) for Java code. These approaches often use superimposition as their composition technique. Following this, the application can then be compiled for the appropriate platform.

To compose variants of GUI documents, we rely on *superimposition* using the approach proposed by Apel and Lengauer (2008). To superimpose GUI documents, each document is represented as trees. These trees mimic the GUI tree of widgets that exist in the runtime system. In each tree, widgets are always parent nodes, with their properties as terminal nodes of the tree. When these trees are composed, nodes are composed from the same level. If the same widget is found in both trees, then the child nodes of both trees are then recursively composed. If a widget node in the refinement is not found in the base program tree, it is copied across with its child nodes to the base program. Widget properties are handled in a similar way in that they are copied to the base tree if they do not exist. If two properties of the same type are found, then the value of the property in the base program is replaced by the property value in the refinement.

When parent nodes are composed, if child nodes are found to be within a `before` hook, we look for the node defined with the hook. If that node is found, all of the nodes within the hook are placed at before that node. If the hook block is specified as an `after` hook, the different nodes contained in the hook are placed after the node specified in the hook declaration.

5.3.2 GUI Document Composition

In Figure 5.6, we illustrate how GUI trees are superimposed, producing a composite tree. With the opening screen of the content store, various buttons are used for adverts and specific content types e.g. video, music etc. In the base document we have several UI elements including the application button, and advertisements. In the Video refinement, we have a button for videos, and a block showing popular movies.

Variant Generation

For runtime adaptation we need to consider a composition approach that can handle dynamic variability of GUI documents. To handle this variability, there are two different approaches that can be used.

The first is to dynamically compose the GUI documents at runtime in the DSPL system. To enable dynamic composition, the base version of the GUI documents,

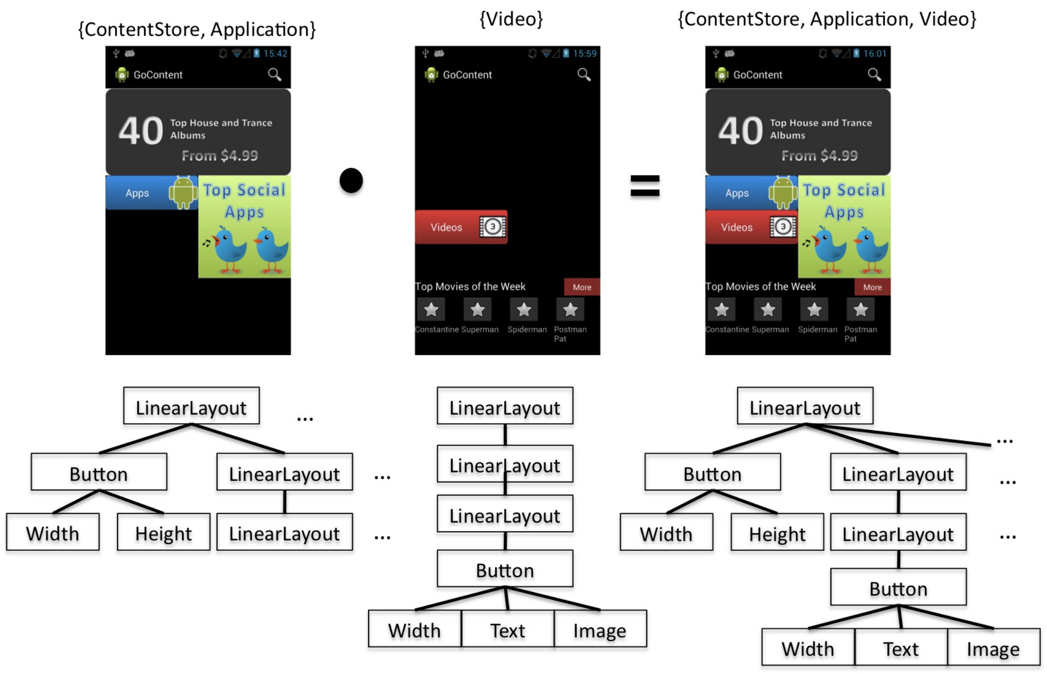

{ContentStore, Application}
{Video}
{ContentStore, Application, Video}
GoContent
40 Top House and Trance Albums
From $4.99
Apps
Top Social Apps
Videos
Top Movies of the Week
More
Constantine Superman Spiderman Postman Pat
LinearLayout
Button
Width
Height
Text
Image
...

Figure 5.6: Composition of Home screen

and refinements need to be deployed with the product. Then depending on the given configuration, the GUI documents are composed at runtime, when they are needed. For runtime composition, all composition tools need to be integrated with the main DSPL application. This approach is often impractical because of constraints of the target platforms. These constraints include the need for GUI document preprocessing tools, which can not always be included in the runtime application, and read/write restrictions for applications.

The second approach is to compose all foreseeable variants at compile time. Then at runtime, when a GUI document is required, the correct variant is chosen, based on the DSPL configuration. By following a compile time approach, no composition and preprocessing tools are required. The approach therefore can be applied more universally, as most platform constraints do not interfere with the approach. We therefore use a compile time approach to dealing with the GUI document runtime variability.

Algorithm 1 Generate all GUI variant configurations

```
Require: A set of relative ordered $Features$
  $Guis \leftarrow$ GETALLGUIREFINEMENTS($Features$)
  for $Gui \in Guis$ do
      $combinations \leftarrow$ GETREFINEMENTCOMBINATIONS($Gui$)
      for $comb \in combinations$ do
          $config \leftarrow$ NEWCONFIGURATION($comb$)
          $config.propagateFeatures()$
          $valid \leftarrow config.isValid()$
          if valid = true then
              ADDCONFIGURATION($Gui$, $Config$)
          end if
      end for
  end for
```

To compose all variants, we first need to generate all needed runtime variants, as depicted in Algorithm 1. When considering variant generation, it is important to produce only unique documents. If variants are generated by every possible configuration of the system, it is likely that there will be many document duplicates. Duplicates can be occur because it is unlikely that a document will be refined in every single feature, leading to more than one feature configuration for a given variant. We avoid this by only generating unique and valid GUI variants.

We therefore find all GUI refinements (line 1). Refinements are found by transversing through each feature module in composition order. Composition order is a relative order that can be set by the user to specify in which order should each feature be composed in a stepwise fashion. This step ends with a two way mapping between which features refine each document. Next, for each GUI document name (line 2), we compute all combinations of features that refine that document (line 3). By carrying this out, we get every unique variant of the document, in terms of features in a configuration.

While all document variants must be unique in their content, they must also be valid in terms of the feature model. Currently, we only compute all combinations of features for each dynamic document. We then need to filter all variants that contain an invalid configuration, in terms of the feature model. For each combination (line 4), we then create a configuration that can be tested for validity. To do this, we initialise a new configuration, manually selecting every feature in the combination (line 5). This is followed by propagating all automatic feature selections (line 6). Automatic feature selections includes feature selections based on either model structural relationships, or using model constraints. An example of this includes selecting any feature parents of manually selected features. The step should end with features being selected explicitly, or implicitly. We then check the given configuration for satisfiability (line 7), using a SAT Solver. The configuration is checked by encoding it into conjunctive normal form (Thüm, 2008; Benavides et al., 2010), creating a SAT problem that can be reasoned over by a off-the-shelf SAT solver, for example SAT4J[1].

5.3.3 Code Generation and Transformation

To support runtime reconfiguration of the GUI at runtime, a number of source transformations and generation need to take place. First we generate the components needed to handle runtime adaptation. The first component is the GUI variant manager, whose purpose is described later in Section 6.5. We generate this component using data returned from the GUI document variant process.

Next, we generate classes to handle runtime feature composition. We adopt this idea from FeatureC++ (Rosenmüller et al., 2008). In FeatureC++, feature classes are generated to handle composition of the different class refinements. We generate feature classes to hold information on what widgets that particular dynamic binding unit affects, and to interface with the underlying language extension for source code composition. A feature class is generated for each dynamic binding unit of the DSPL,

[1]http://www.sat4j.org/

which contains the following methods:

- `getAllAlteredWidgets()`. This method returns a collection of all the widget identifiers that are to be adapted by the dynamic binding unit.
- `addSourceRefinements()`. This method is used to compose the current system with the source code refinements. We can do this by making the appropriate calls to the underlying language API or language keywords.
- `removeSourceRefinements()`. This method is used to to remove the current refinements from the current system. Just like the previous method we make the appropriate calls to the underlying language API or language keywords.

Following the generation of these components, source transformations are needed. First, for all classes that require GUI adaptation while the GUI is active, declared earlier, additional methods are required. We generate a method to handle the adaptation of every GUI document which that class uses, which we describe in more detail in Section 6.4.1. Each of these methods handles the inflation and composition for a particular GUI document.

Also references to GUI documents need to be updated so that the correct variant of that document is used. For example in the Android platform, different methods including `setContentView` and `inflate` use the GUI document as a parameter. The `inflate` method takes a GUI document, parses it, and returns a GUI tree that can be applied to the screen. If the `setContentView` method is used, this inflates and applies the GUI tree to the screen automatically. These methods therefore need the correct document variant when they are invoked. If these method calls are found within classes or class refinements, they are altered to get the correct variant first, before then parsing it to the invoking method, as shown in Listing 5.6.

Listing 5.6: Code transformation of an Android method

```
//Original Implementation of onCreate() method
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ....
}

//Implementation after transformation
DSPLResourceGetter dsplrg = (DSPLApp)getApplicationContext(). getDSPLRG();
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(dsplrg.getResourceVariant(R.layout.activity_main));
        ...
}
```

5.3.4 Language Preprocessor/Compiler

Language approaches including FOP allow for modularisation of adaptation, which is then preprocessed at compile time. Depending on the language approach used, different transformations are carried out. For example with FeatureC++, class refinements for runtime adaptation are transformed using the *decorator pattern*.

We, therefore, following the code transformations proposed earlier, preprocess and compile the source code using the appropriate tools, to get the final DSPL product.

5.4 Summary

This chapter presented an approach for designing and implementing a DSPL that includes GUI variability. We have shown how the application and context can be modelled using a single modelling notation. Next we described how variability in GUI documents can be realised allowing for static and runtime adaptation. This approach requires different derivation processes to handle these two adaptation strategies.

In the following chapter, we describe how the DSPL can continue to modify itself at runtime, using the assets created in the domain engineering phase of developement. Attention is paid to the centralised management system, and the different components that assist in adaptation of the GUI.

6 Runtime Phase

Contents

6.1 Introduction

So far we have discussed the different development phases involved with the SPL. Unlike a conventional SPL, a DSPL continues to reconfigure and adapt at runtime. This adaptation can be caused by manual feature selection, or by context changes. We therefore require a management system to manage the different context events, and to handle the automated system reconfiguration. Because it is possible to have more than a single mobile DSPL running simultaneously, it would be beneficial if such a system was not packaged with each application. By having a single DSPL management system on the device, we can get greater runtime reuse of this component.

The rest of this chapter is organised as follows. First, we describe the architecture of DSPL application, which is an output of the activities carried out in the last chapter in Section 6.2. Next in Section 6.3, we present the DSPL middleware designed to manage the contexts and configurations of each DSPL. Following this, in Section 6.4, we describe the GUI Adaptation Manager component, responsible for managing the

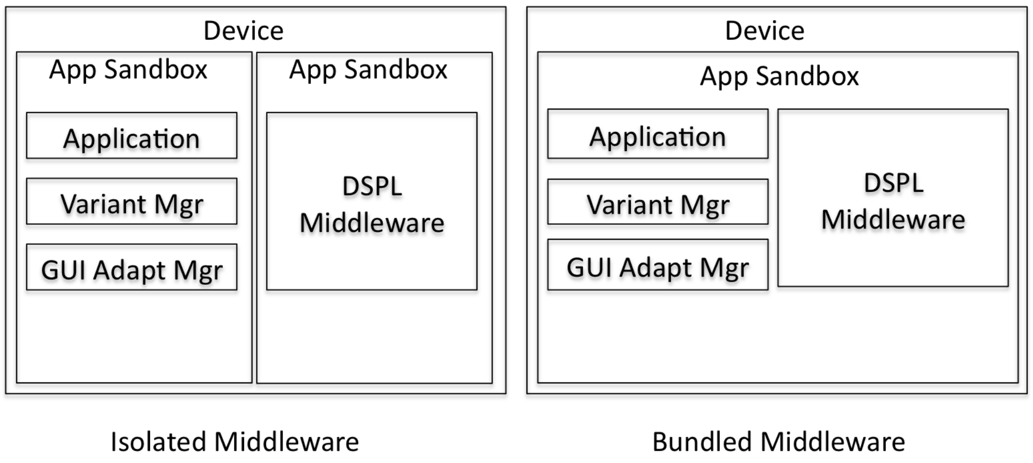


Figure 6.1: Application Architectures

adaptation of the GUI and its business logic. Next in Section 6.5, a description of the GUI Variant Manager component is presented. This chapter is finally summarised in Section 6.6.

6.2 Application Architecture

Following the derivation process, a mobile DSPL is produced. The main components of the application are the following:

- **DSPL Middleware** named *FeatureDroid* facilitates context acquisition, and product configuration management.
- **Application Source** is the main source code of the application, that can be adapted based on the changing product configuration.
- **GUI Adaptation Manager** invokes the adaptation in the GUI when a reconfiguration takes place.
- **GUI Variant Manager** provides the ability to manage what GUI document variants are appropriate for the current configuration.

Two different architectures are depicted in Figure 6.1. These two architectures illustrate the two different usage scenarios of the DSPL middleware. The first architecture illustrates the middleware being used as a separate entity, running isolated from the DSPL application. By running the middleware in isolation to the application, the middleware can provide the same support to multiple running DSPL applications on the

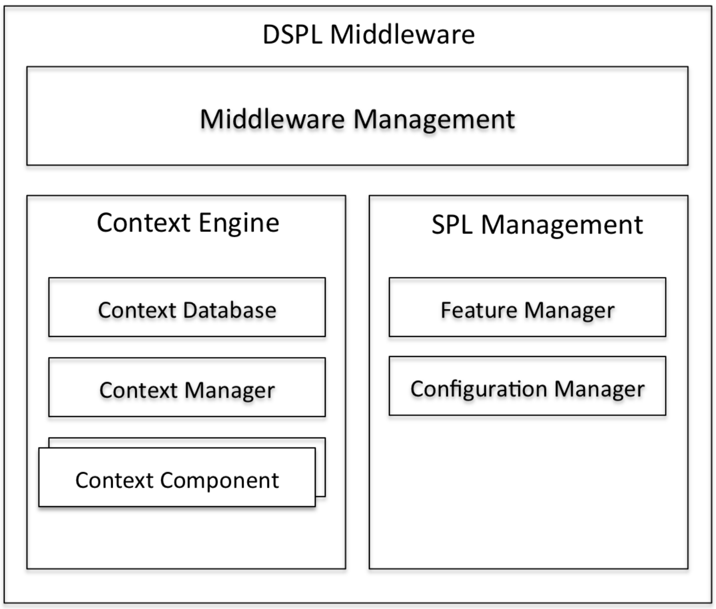


Figure 6.2: FeatureDroid Architecture

device. This can improve runtime context reuse, as contexts can be used to monitor for multiple applications. It also means that on mobile platforms where applications have set memory limits, the memory required for the middleware can be removed from the DSPL application. This can also save memory on the device, as multiple instances of the middleware are therefore not required.

The second architecture depicts the DSPL application being completely standalone. This scenario entails that the DSPL middleware is bundled with the main application and the other components. In this approach, this can allow for greater security, as all product feature and configuration data, along with context components can be part of the same application sandbox as the main application.

6.3 FeatureDroid

As part a DSPL, runtime management code is required for managing the features and contexts at runtime. In this section, we discuss our proposed DSPL middleware, named FeatureDroid designed to carry out this role.

6.3.1 Middleware Architecture

The middleware can be seen to be made up of three main components as shown in Figure 6.2 including:

- **Context-Acquisition Engine**. This component primarily controls and handles all acquisition of context data, reasoning for higher level context information, and context component life cycles.
- **SPL Management**. This component controls and handles the main SPL management. This includes feature management for the different running feature models, and their configurations.
- **Middleware Management**. This component brings the two previous components together. It also acts as the agent for providing communication between the other two components, and also the running application using the middleware.

6.3.2 Context-Acquisition Engine

Context can be used to monitor the current situation of the user and/or device, and then alter the DSPL configuration. To monitor each context, a context management system is required. Here, we introduce the context management component of the DSPL middleware.

Context Component

A context component is self-contained, and manages a particular context. Context components are implementations of the atomic contexts defined in the context model, back in Section 5.2.2. Each context component has its own lifecycle that is managed by the context manager.

In Figure 6.3, we outline the structure of a context component. Each context component is described with a name. The name of the context component should match the context it implements in the context model. Context components are designed to allow for different context value definitions in context models. We enable this ability by the use of ContextValueSets. A ContextValueSet is a collection of values that are used to infer particular contexts. Each application using the context engine has its own ContextValueSet, allowing multiple context definitions to be handled by a single

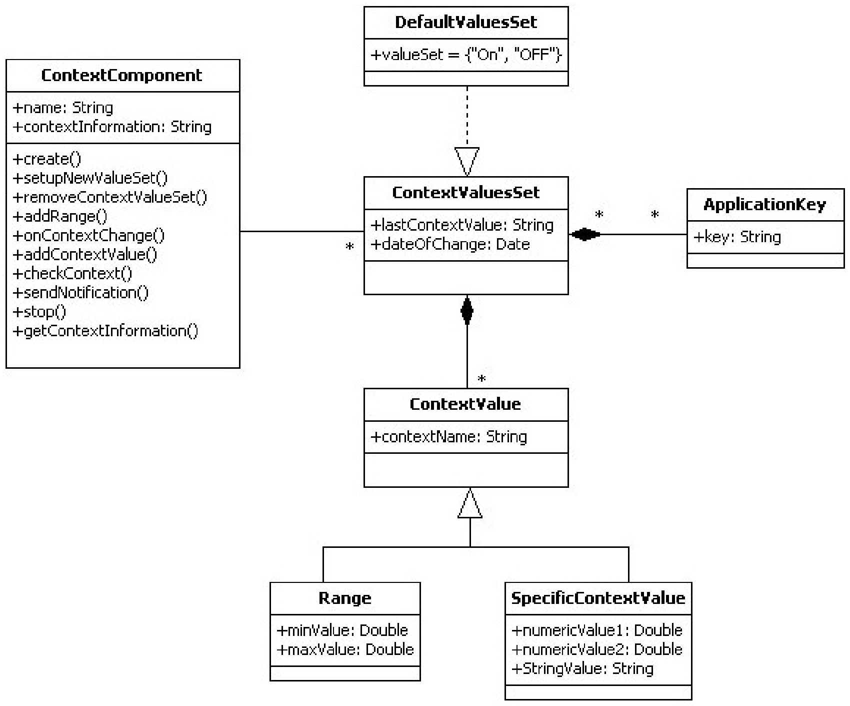


Figure 6.3: A Context Component

instance of that context component. By using context components, more meaningful context information can be inferred from raw context data. By inferring higher level information about the context, we need not inform the application about every change, but only relevant ones. For example, battery level in devices is numerical. It is probable that not every change in percentage is required by larger changes for example, if the battery level is high, medium, or low. This type of data is handled by ranges. Each ContextValueSet has an ApplicationKey, which acts as an identifier to what applications do to these context values below. This then allows each application to define its own context values, which can all be reasoned over together.

Context Engine Manager

Context components are managed by a centralised context manager, as shown in Figure 6.4. All context component life cycles are managed by the manager to suit the needs of all applications using contextual information. The ContextManager is designed to handle static and dynamic context deployment, and to handle all communications back to applications subscribed to context updates.

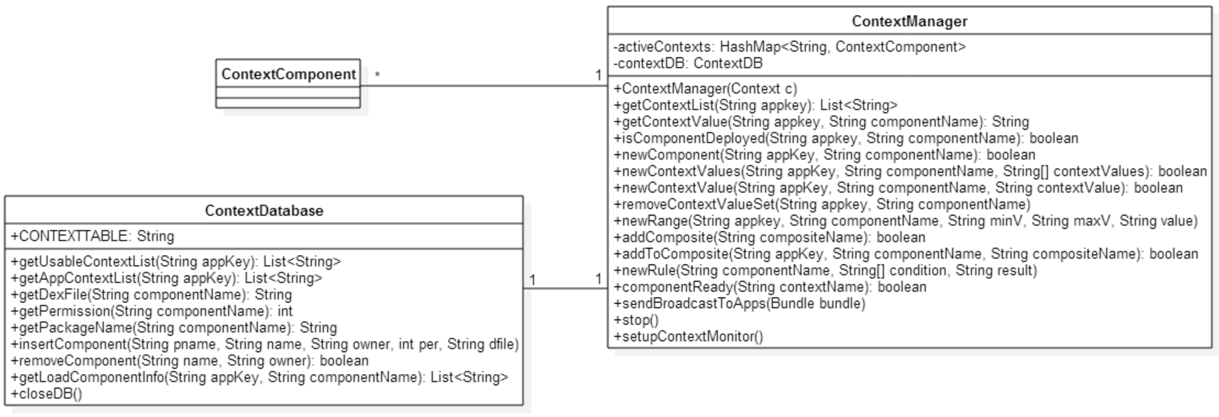

ContextComponent
*
1
ContextManager
-activeContexts: HashMap<String, ContextComponent>
-contextDB: ContextDB
+ContextManager(Context c)
+getContextList(String appkey): List<String>
+getContextValue(String appkey, String componentName): String
+isComponentDeployed(String appkey, String componentName): boolean
+newComponent(String appKey, String componentName): boolean
+newContextValues(String appKey, String componentName, String[] contextValues): boolean
+newContextValue(String appKey, String componentName, String contextValue): boolean
+removeContextValueSet(String appkey, String componentName)
+newRange(String appkey, String componentName, String minV, String maxV, String value)
+addComposite(String compositeName): boolean
+addToComposite(String appKey, String componentName, String compositeName): boolean
+newRule(String componentName, String[] condition, String result)
+componentReady(String contextName): boolean
+sendBroadcastToApps(Bundle bundle)
+stop()
+setupContextMonitor()
1
1
ContextDatabase
+CONTEXTTABLE: String
+getUsableContextList(String appKey): List<String>
+getAppContextList(String appKey): List<String>
+getDexFile(String componentName): String
+getPermission(String componentName): int
+getPackageName(String componentName): String
+insertComponent(String pname, String name, String owner, int per, String dfile)
+removeComponent(String name, String owner): boolean
+getLoadComponentInfo(String appKey, String componentName): List<String>
+closeDB()

Figure 6.4: Context Engine Manager

Context Component Database

The context acquisition system should be able to handle many different context components deployed from different applications. With the context components, some deployed components may be intended to be usable by other applications on the device, while some may be intended for private use only. It is also possible that different context components are implemented in different package names, which need to be known for dynamic loading. To keep track of this information, a database all contexts that are available is needed, which can seen in Figure 6.4. In the database the following information is stored for each context:

- **id**. A unique identifier for the context component, stored as an integer.
- **name**. This is the name of the context component, used in the class definition of the component, stored as a String. An example of this include ”BatteryContext”, and this should match the name of the actual implementation class.
- **owner**. The owner of the context component is the application that deployed the context component. This is stored as a String. The owner identifier specified should be identical to what is used in the DSPL instance or with the context engine separately.
- **permission**. The permission of the context component corresponds to whether the context component is private or public. By a context component being public, it can be used by any application id, compared with just the owner being able to use a private context component. This is stored as an integer, where zero equates to a public context, and one equates to a private context component.

- **dex_file**. The dex file is the compiled unit that contains the context component. Because our implementation is based in Android, context components are compiled to .dex files. Using other platforms and languages, these units could be dynamically bound libraries for example Jar, DDL, or SO. Multiple context components can be contained in a single compiled unit, or in their own units.

After the database is populated, it can be queried by the Context Engine Manager. Queries to the database can be for internal and external use. External queries include giving the end applications the ability to request which contexts are currently available for use, and which contexts are owned by that application. Internal queries are used by the manager to request the information required to dynamically load and instantiate the different context components. This is carried out by invoking the method *getLoadComponentInfo(String, String)*. This method returns all data required to first check that the context can be used by the requesting application according to its permission settings. If the requesting application can use the context component, the data returned is then used by the dexloader and classloader.

Context Deployment

Deployment of context components can be carried out either at compile time or at runtime. Compile time deployment is carried out in the situation of the packaging the middleware with a single application. When compile time deployment is handled, the only task needed is that all context components are added to the database when the application is first used.

Because of the many different context components that may be needed by applications, it is likely not all contexts that may be required in the future by an application will be deployed with the middleware. Furthermore, some applications may require specialised context components that interact with online services, for example, social networks. Therefore, by deploying context components at runtime, each application can add the context components that it needs, which can then be shared with other applications as shown in Figure 6.5. As mobile devices typically run applications within their own sandboxes with private storage, for security reasons, it is therefore prohibited to deploy executable code into an arbitrary application. Also when using platforms such as Android whereby applications are compiled to a single executable by default, runtime deployment is even harder.

To deploy context components at runtime, first at compile-time, the components are

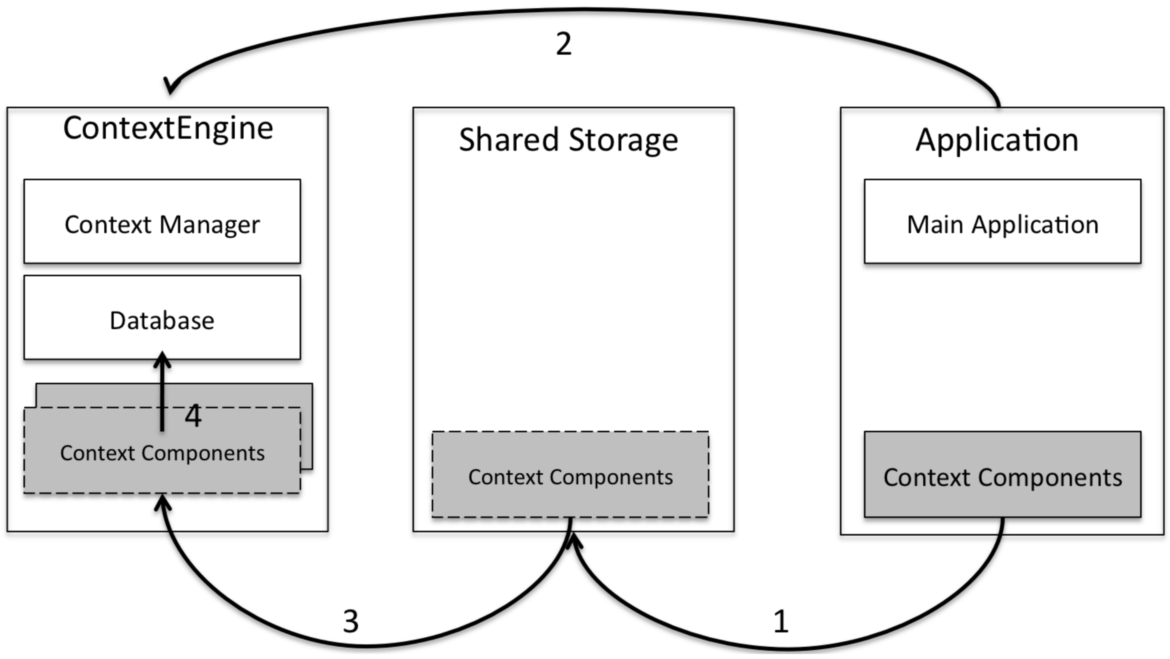


Figure 6.5: Runtime Deployment of Context Components

separated, and compiled into their own executable. This executable is the packaged with the rest of the application as a separate library. Then at runtime, because there are restrictions of inserting executable code into another application, the components are copied to a mutually shared storage space, for example, a SD card (1). Once the executable code is copied over, the application can call the middleware to deploy the executable, with its temporary location (2). Next, the middleware can then copy the executable into its own private storage for execution (3). Finally, the component is then added to its database containing information on its owner, and usage permissions (4).

Context Use

Once a context component has been deployed to the middleware, it then can be used to form the running context model of the system using methods in the ContextEngine-Manager class, shown in Figure 6.4. Contexts are started when context values are added by an application e.g. adding a LOW range to represent a battery level of 0% to 20%. This can be handled using the methods *newContextValue, newContextValues*, and *newRange* in the ContextEngineManager. If the context component is already running, then the engine attempts to add the new context values to that component for reasoning. If the component is deemed a private context component for a different application, then those context values are not added.

If the context needs to be started for the new context values to be added, then firstly, like a running context component, the permissions of that component are checked. If the component can be used by that application, then the location of the executable code, and component package/namespace are queried by the database. These details are then used by the class loader to load the context component, which then is added to a container of running contexts.

When contexts are no longer needed by applications, the applications cannot just stop the context component from running. This is because the context component may also be needed by other applications. Allowing a single application to stop context components would lead to unpredictable behaviour of other applications. Because of this, instead, we allow an application to inform the context manager that it does not require the use of a context anymore. Like an application can add contexts by adding context values, removing contexts is handled by removing context values. When context values are removed, the manager first checks if the values exists for that component. If the values are currently running on the component, it is consequently removed. Following this, the manager checks if that context component is needed by other applications. This is handled by checking the number of context value sets that exist in the context component. Based on this, if the number of ContextValueSets is zero, the component is no longer needed, and is therefore stopped, and removed from the running lists of components.

Context Checking

Each context component is responsible for checking and reasoning over its own context data. As described earlier, each application can use a context component by specifying their own higher level context values. This checking can be handled when new low level context data is acquired, for example when a light sensor receives a new value for the number of lumens it senses. This new low level data is then checked by firstly updating the current low level context data value. The new context data is then checked over all ContextValueSets to check for newly inferred higher level context values.

To check for new context values using the higher level context information is inferred differently, depending on the type of low level context data. For ranges, this is handled by checking the low level value against the predefined lower and higher bounds of a

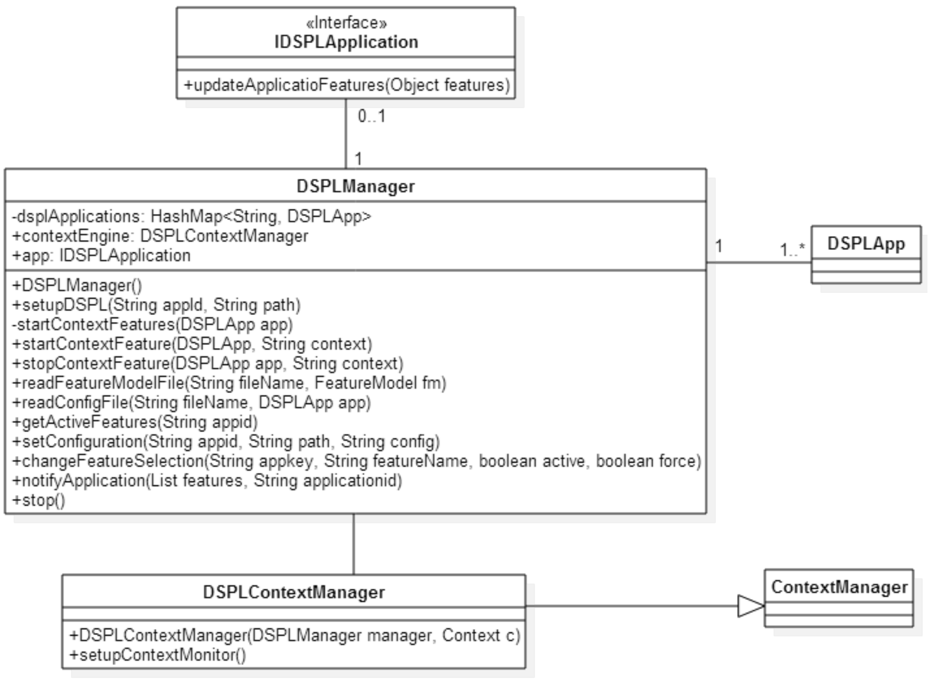

«Interface»
IDSPLApplication
+updateApplicatioFeatures(Object features)
0..1
1
DSPLManager
-dsplApplications: HashMap<String, DSPLApp>
+contextEngine: DSPLContextManager
+app: IDSPLApplication
+DSPLManager()
+setupDSPL(String appId, String path)
-startContextFeatures(DSPLApp app)
+startContextFeature(DSPLApp, String context)
+stopContextFeature(DSPLApp app, String context)
+readFeatureModelFile(String fileName, FeatureModel fm)
+readConfigFile(String fileName, DSPLApp app)
+getActiveFeatures(String appid)
+setConfiguration(String appid, String path, String config)
+changeFeatureSelection(String appkey, String featureName, boolean active, boolean force)
+notifyApplication(List features, String applicationid)
+stop()
1
1..*
DSPLApp
DSPLContextManager
+DSPLContextManager(DSPLManager manager, Context c)
+setupContextMonitor()
ContextManager

Figure 6.6: The DSPL Manager

range. If a value is found within a particular range, that context information, is then returned.

Finally, if the new context information is different from the currently set value, it is therefore updated, and the component broadcasts to the context engine the new context value with the application id that the context value is related to.

6.3.3 DSPL Management

The second part of the middleware includes the components for handling the feature models, and configurations of the DSPL instances. In Figure 6.6, we show the structure of the DSPL Manager, which carries out all centralised management tasks. The DSPL Manager has a ContextManager associated with it. As a method of decreasing coupling between the context-acquisition engine and the DSPL manager, we create an extended class called DSPLContextManager. In this extended class, instead of broadcasting context changes in the *setupContextMonitor* method, we update the feature configuration of the application in question.

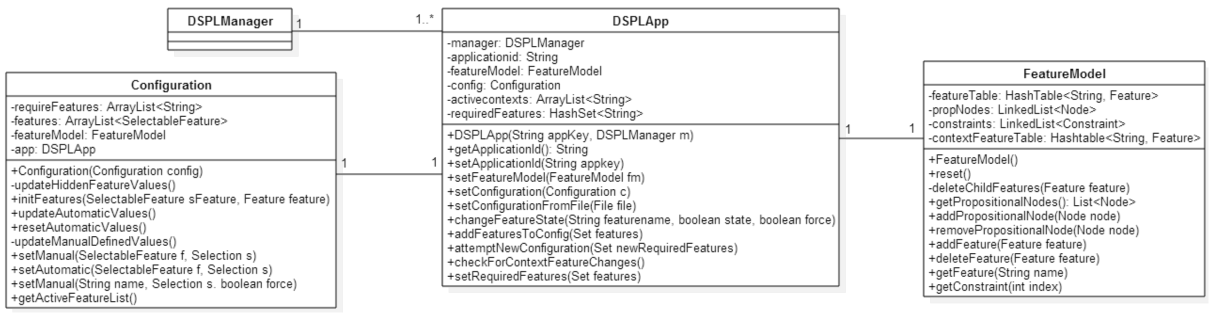

DSPLManager
1
1..*
DSPLApp
-manager: DSPLManager
-applicationid: String
-featureModel: FeatureModel
-config: Configuration
-activecontexts: ArrayList<String>
-requiredFeatures: HashSet<String>
+DSPLApp(String appKey, DSPLManager m)
+getApplicationId(): String
+setApplicationId(String appkey)
+setFeatureModel(FeatureModel fm)
+setConfiguration(Configuration c)
+setConfigurationFromFile(File file)
+changeFeatureState(String featurename, boolean state, boolean force)
+addFeaturesToConfig(Set features)
+attemptNewConfiguration(Set newRequiredFeatures)
+checkForContextFeatureChanges()
+setRequiredFeatures(Set features)
Configuration
-requireFeatures: ArrayList<String>
-features: ArrayList<SelectableFeature>
-featureModel: FeatureModel
-app: DSPLApp
+Configuration(Configuration config)
-updateHiddenFeatureValues()
+initFeatures(SelectableFeature sFeature, Feature feature)
+updateAutomaticValues()
+resetAutomaticValues()
-updateManualDefinedValues()
+setManual(SelectableFeature f, Selection s)
+setAutomatic(SelectableFeature f, Selection s)
+setManual(String name, Selection s. boolean force)
+getActiveFeatureList()
1
1
1
1
FeatureModel
-featureTable: HashTable<String, Feature>
-propNodes: LinkedList<Node>
-constraints: LinkedList<Constraint>
-contextFeatureTable: Hashtable<String, Feature>
+FeatureModel()
+reset()
-deleteChildFeatures(Feature feature)
+getPropositionalNodes(): List<Node>
+addPropositionalNode(Node node)
+removePropositionalNode(Node node)
+addFeature(Feature feature)
+deleteFeature(Feature feature)
+getFeature(String name)
+getConstraint(int index)

Figure 6.7: A DSPL Application Instance

DSPL Instance

As this middleware is designed to handle multiple applications, it needs to handle multiple DSPL instances. Therefore each DSPL application running has its DSPL instance associated with it, shown in Figure 6.7. These instances are self contained units, that each manage the configuration of their associated application. These instances then can send reconfiguration notifications to the application through the middleware. Each running DSPL instance has a unique application identifier, a feature model, and a configuration associated with it. Furthermore, we store separately a list of what contexts are currently being monitored as part of the context acquisition. When applications make interface calls to the middleware, the application identifier is required to ensure the correct instance is manipulated. At runtime, each DSPLApp class instance represents a DSPL application running on the mobile device. Context updates from the context-acquisition are sent to each DSPLApp instance through the DSPLManager.

Instance Initialisation

As described earlier, each DSPL instance has a feature model, and a configuration. These files are bundled with each DSPL application at compile time. Depending on the usage scenario of the middleware, the files are read from different locations. If the middleware is bundled with the application, the middleware can read the files directly from the application bundle. On Android, these files can be stored in the application assets folder. If the middleware is used externally from the application, these files then cannot not be read directly from the application. Instead, they need to be moved to a mutually readable storage area, for example an SD card on a device.

Once in a readable location, the application can call the middleware to read the files and add them to a new DSPL instance. This then loads the feature model, and reads the initial configuration of the system. This is then followed by its first reconfiguration using the features in the configuration.

Reconfiguration

When a context event is acquired in a specific context, the context manager gets the new context values. Depending on what context values have been changed, the appropriate features in the context model representing context values are either added or removed from the running configuration.

When a feature is required to be active. Firstly, the engine checks if a reconfiguration is required. This is carried out by checking all currently active features. This is because a feature may already be active, not directly because of a requirement, but indirectly due to a feature model constraint. If the feature is already active indirectly, the feature is just added to the list of required features, with no reconfiguration needed.

If that feature is not already active, the list of required features is updated, and a reconfiguration is required. A reconfiguration firstly begins with the current configuration being reset, and all required active features being added to the configuration.

Next, after all selected features are added to the configuration, other features that require being selected automatically because of a feature relationship, or constraint are selected. An example of such automatic selection includes the selection of all parent features from a selected feature back to the root. This is because all subfeatures included in a configuration must have its parent.

Finally, the configuration is analysed for its satisfiability. This check is carried out by a SAT solver by encoding the configuration into conjunctive normal form (Thüm, 2008; Benavides et al., 2010) which can be reasoned over, using standard off the shelf SAT solver like SAT4J[1].

If the new configuration is tested to be valid, then it is added to the DSPLApp as the current configuration, with the old being discarded.

Then depending on changes of which context features are active/inactive, the middleware will either need to add another context to the running application or remove a running context. Additional contexts are added by requesting new context values to be monitored by the context management. The context management then will either start a new context component and add the new context values, if one is inactive, or will add the additional context values if the component is already running. Contexts

[1]http://www.sat4j.com

that need removing are removed by requesting the contexts values to be removed by the context management.

Reconfiguration Notification

When there is a reconfiguration, the application is notified of the change that needs to be carried out.

Because of the two different usage scenarios of the middleware, there are two different approaches in which the DSPL applications can be notified of this change.

The first method of notification can be handled using an object callback. Using this method of notification is only possible when the middleware is packaged with a DSPL application, running inside of a single sandbox. This approach is handled by the use of an Interface, shown in Listings 6.1, that is implemented in a specific class of the application to receive the feature changes. When a reconfiguration then takes place, this method is called, passing the list of features that are now active back to the application. Following this, the application can make the approach reconfiguration based on this new configuration.

Listing 6.1: Interface for reconfiguration call backs

```java
public interface IDSPLApplication {
        public void updateApplicationFeatures(Object[] features);
}
```

The second method of notification can be handled via interprocess communication (IPC). This method is used when the middleware is used centrally outside of any single DSPL application, and is used as a method of communication between application sandboxes. For this approach, the application registers and listens for notifications from the middleware fitting a particular signature.

Once the application has received the notification of the configuration change, different changes to the application can commence.

6.4 GUI Adaptation Manager

In this section, we describe the GUI adaptation manager, which orchestrates the adaptation of the application and its GUI according to the new configuration.

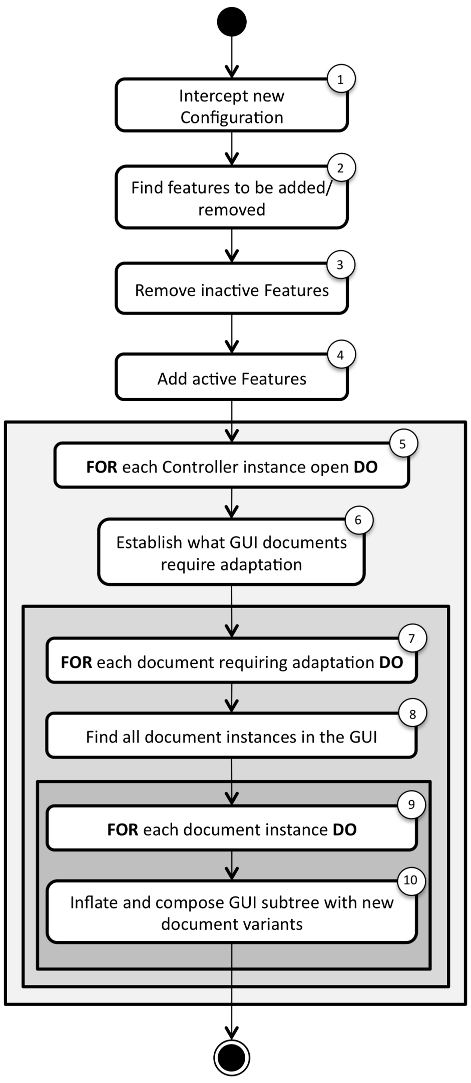

1
Intercept new Configuration
2
Find features to be added/ removed
3
Remove inactive Features
4
Add active Features
5
FOR each Controller instance open **DO**
6
Establish what GUI documents require adaptation
7
FOR each document requiring adaptation **DO**
8
Find all document instances in the GUI
9
FOR each document instance **DO**
10
Inflate and compose GUI subtree with new document variants

Figure 6.8: Reconfiguration Process

6.4.1 Reconfiguration

Reconfiguration of the DSPL is carried out by the GUI Adaptation manager. This component handles the visual adaptation of the GUI, and other elements that have been defined in source code that may require reconfiguration, for example, gestures. The overall process of reconfiguration is depicted in Figure 6.8. The component intercepts the reconfiguration notifications from the DSPL middleware using IPC (1).

Firstly, the manager needs to be aware of what dynamic binding units are now active/inactive (2). To do this, the manager checks the new configuration against the last configuration. This activity allows us to know what features need to be added and removed from the running system, which can include sourcecode and GUI refinements.

As described earlier, for each dynamic binding unit, or compound feature present in the compile application, a *Feature* class is generated. Firstly, dynamic binding units that are no longer active in the configuration are dealt with (3). This begins by the component first calling the feature class for all the widget names it adapts. The returned set of widgets are then added to a global list of altered widgets for that adaptation transaction. Next, all source adaptations associated with a dynamic binding unit are removed. These adaptation are removed by making the appropriate calls to the language meta-level program, or language specific reserved terms.

Secondly, when a new configuration requires a previously inactive dynamic binding unit to be active, firstly that feature class is instantiated (4). Following this, like the last step, all widgets names associated with that dynamic binding unit are added to the global list. Lastly, the feature class, as in the last step make specific calls to the language meta-level program, or language keywords to activate the source code adaptations.

After all the widget names that have to be adapted in the reconfiguration have been collected, adaptation of the GUI can take place. To do this, the GUI manager iterates over all GUI controller instances currently open, calling each to reconfigure parsing the set of altered widgets (5). This set of altered widgets can be just concerning a single GUI document for a single controller, or can even effect multiple GUI documents across multiple controllers. As discussed earlier, single GUIs can be made up of more than a single GUI document. For this reason, the controller needs to distinguish firstly what widgets are associated with that controller, and what GUI documents are affected in the adaptation (6).

In each generated controller, a generated map of widgets that can be altered is stored. This map also references which GUI document they are associated with. This allows the controller to firstly remove from the set the widgets that are not related to it. It also then allows the controller to identify which GUI documents need to be reloaded. The controller handles this by sequentially looking up every widget in that set of altered widgets. If the widget name is not found in the map, it can be removed from the set. But, if the widget is found, that widget is retained and the GUI document it is associated with is returned. This is then used to call the appropriate reconfigure method, generated for each GUI document in the application engineering phase of the development (7).

GUI documents can be used to represent portions of the GUI. This allows for the same layout and styling to be applied to multiple parts of the GUI without the need for duplication, for example with lists. This means that in a GUI, there can be multiple instances of a single GUI document. As a consequence, we need to make sure that every instance of the GUI document in the GUI tree is adapted. When the specific reconfigure method for a GUI document is executed, we first get all instances of that GUI document (8). This is carried out by transversing the GUI tree, searching for widget identifiers that match the root node in that GUI document. The root of each GUI document is gathered during the product derivation stage described in the last chapter.

Once all instance GUI subtrees have been found, for each instance, a new variant of the GUI document is inflated, and composed with the GUI subtree (10).

GUI Tree Composition

Composition of two GUI subtrees is shown in Algorithm 2. This method recursively composes each GUI subtree. As described earlier, a list of widgets that require alteration is created by the GUI manager when each source code refinement is applied. For each of the widgets that are in the list, we check if that widget is in the new GUI tree. If the widget does not exist in the new tree, we can safely assume that the widget needs removing from the current GUI tree. If the widget is in the new tree, it needs to be moved from the new tree, to the current tree. Before this can happen, it needs to be checked if itself is a container of other widgets. In the event that it is, the set of child widgets from both trees need to be composed. This is carried out by checking if any of the child widgets of the current widget are those that need altering. If a child does not require alteration, the recursive method is called, with the child, its parent, and the new widget as input. However, if that child does require alteration, it is therefore not moved, and that widget can be removed from the list of altered widgets, followed by transfer of state from the old child to the new child.

State transfer is handled by a component generated in the design phase, using the state transfer templates. To copy the state from one widget to another, the two widgets are checked if they are the same type. Following this, depending on the old widget's object type, the corresponding generated method is called. Each generated method then contains the transformed source code defined in each state transfer template.

Algorithm 2 GUI Tree Composition

```
Require: A set of altered widgets $Widgets$, the current GUI tree $curTree$, and the new
    GUI tree $newTree$
 1: for $Widget \in Widgets$ do
 2:     procedure MOVEWIDGET($Widget, curTree, newTree$)
 3:         if $Widget \in newTree$ then
 4:             $newWidget \leftarrow newTree.get(Widget)$
 5:             $parent \leftarrow newWidget.getParent()$
 6:             $oldParent \leftarrow curTree.get(parent)$
 7:             if $Widget \in curTree$ then
 8:                 $curWidget \leftarrow curTree.get(Widget)$
 9:                 if $curWidget.children \neq \emptyset$ then
10:                     for $Child \in curWidget$ do
11:                         if $Child \notin Widgets$ then
12:                             MOVEWIDGET($Child$, $newWidget$, $curWidget$)
13:                         else
14:                             $Widgets.remove(Child)$
15:                             if $Child \in newWidget$ then
16:                                 $newChild \leftarrow newWidget.get(Child)$
17:                                 TRANSFERSTATE($Child$, $newChild$)
18:                             end if
19:                         end if
20:                     end for
21:                     $position \leftarrow oldParent.indexOf(curWidget)$
22:                     TRANSFERSTATE($curWidget$, $newWidget$)
23:                 else
24:                     $position \leftarrow parent.indexOf(newWidget)$
25:                 end if
26:                 $curParent.addWidget(newWidget, position)$
27:             end if
28:         else
29:             $curTree.remove(Widget)$
30:         end if
31:     end procedure
32: end for
```

Other GUI Adaptations

Following adaptation of the visual properties of the GUI, adaptations defined in the `onGUIConfiguration` methods are executed. These methods are executed in order of feature composition order, whereby the base and each refinement is executed before the next. As discussed in the previous chapter, the properties that may be adapted need to be set to their default values in the base version of the class. Following this, each refinement is executed, updating the properties of the GUI according.

6.5 GUI Variant Manager

The GUI adaptation manager component does not function independently, but relies on a separate component designed to manage the different GUI document variants. This variant manager does not manage variants in terms of files or physical resources, but by references. These references can then be used much the same way static GUI documents are used in the rest of the system.

The variant manager is generated automatically in the application engineering phase presented earlier, using output data created when generating each GUI document variant. Within the component, for every GUI document that contains runtime variability, there are the following data structures:

- **Feature-Variant Map.** Because we generate all valid unique variants for each GUI document requiring runtime adaptation, we therefore need to know what variants are needed for a given configuration. This structure is a key-value structure, with the collection of active features as the key, and the variant reference as the value. To avoid duplicate variants across different configurations, each key only contains the active features from a configuration which actually adapt that GUI document. An example of this map is depicted in Figure 6.9. This map is for the main home screen of the scenario application in this thesis.

- **Feature Array.** This array compliments the feature-variant map, by listing what features are associated with a particular GUI document. During variant retrieval, this list is used to remove unrelated features from the key looked up in the feature-variant map.

With these data structures also includes a generated method designed to use the correct GUI document specific feature-variant map, and feature array during variant

Active Features	Variant
Applications	main_screen_1
Applications,Music	main_screen_2
Applications,Video	main_screen_3
Applications,Music,Video	main_screen_4

Figure 6.9: Map of features and main screen document variants

retrieval, which is explained next.

6.5.1 Variant Retrieval

When a GUI document variant is needed, in the case of GUI adaptation, or when the GUI is first inflated, it is requested via a single method. This entry point takes the name of the GUI document in question, and then invokes the specialised method needed for that GUI document name. If the GUI document for that name has only a single variant, and is therefore not managed by the manager, the original reference is returned and can be used as before.

If a specialised method is invoked, first we copy the list of active features in the current configuration. Next, all features not associated with the GUI document in question are removed from that list. To remove the unassociated features, we use the GUI document feature array, and remove features from the configuration list that are not in the feature array. Next, we sort the features into alphabetical order, and output the list as a single string. But ensuring the different features are in alphabetical order, we ensure that a single configuration can only produce a single string. Lastly, a map lookup using the string of features as the key, with the resulting GUI document reference being returned.

6.6 Summary

In this chapter, a description of the compile application architecture, the supporting middleware, and components needed for GUI adaptation. This chapter concludes the final half of the approach this dissertations proposes. Following this is the validation part of the thesis, where validate and evaluated the proposed approach.

Part III

Validation

7 Implementation

Contents

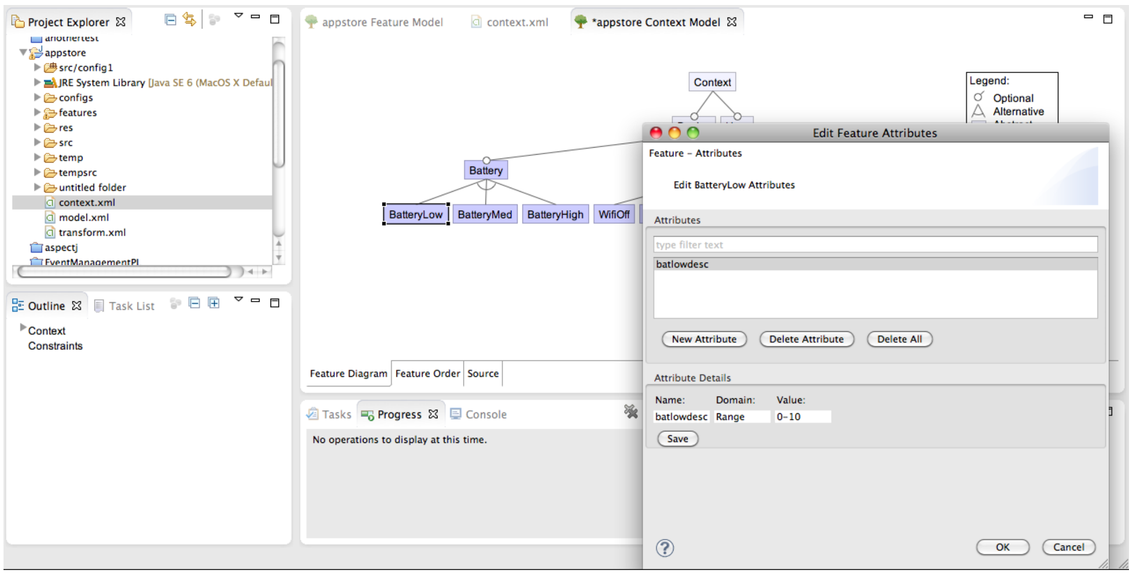
7.1 Introduction

In Chapters 5 & 6, we described the development activities, and components proposed to enable compile time and runtime GUI adaptation in DSPLs. As part of our validation, tool support, and supporting middleware prototypes were implemented. These prototypes were developed primarily for the Android platform, and for the Eclipse IDE. In this chapter, we describe more of the technical details of the prototypes.

This chapter is organised as follows: In Section 7.2, we present the tool support developed to assist the developer while undertaking the main engineering tasks of the SPLs, including modelling, implementation, and composition. Then in Section 7.3, we present the supporting middleware developed to take care of context management, and configuration management. Finally, in Section 7.4 we summarise the chapter.

7.2 Design Phase Tools

Tool support was implemented on top of FeatureIDE (Kastner et al., 2009) to handle DSPLs for the Android platform. In this tool, we implemented components to handle



Name: batlowdesc Domain: Range Value: 0-10

Figure 7.1: FeatureIDE Feature Attribute Extension

variant generation and code transformation. Currently the tool support is design to handle Android GUI layout documents. In Android, other XML documents can be used e.g for defining list menus, application wide styles, and language related text. It is possible that the tool support can be extended to handle these types of documents, but because each document type is handled different, supporting logic will be required for dynamic adaptation.

The first component extended was the feature modelling tool.

7.2.1 Feature Modelling

To assist in feature modelling, we developed additional tool support for our type of feature models in FeatureIDE. This firstly included the ability to add feature attributes as shown in Figure 7.1. Each feature can contain any number of feature attributes, made up of a name, domain, and value.

Second, the ability to model the context, and the context rules was added. To model the different aspects, each FeatureIDE project is now given three feature model files, one for the main application, one for the context model, and an automatically aggregated feature model used for defining context rules. By separating each part into different models, the context model can then be reused over possibly many different DSPL projects.

7.2.2 GUI Document Composition

GUI document composition was implemented on top of the general purpose composition tool, FeatureHouse[1] (Apel, Kastner and Lengauer, 2009). In FeatureHouse, tree composition is carried out by superimposing FSTTree nodes, both nonterminal and terminal. To enable the composition of Android GUI documents, different XML elements have to be mapped to these FSTNodes. In our implementation, all XML nodes in the tree are mapped to FSTNonTerminal nodes. The attributes within each XML node however are mapped to FSTTerminal nodes. During composition, when two terminal nodes of the name identifier are found, we replace the base attribute with the new value.

To extend FeatureHouse with our implementation, there are two main packages that need to be extended, *builder* and *printer*. The builder package contains the logic used to used to process each source file with a particular filename suffix. Most builders then reference the correct language parser to parse that source code into FSTNodes. In our extension, we have our XML parser, which parses each XML document to FSTNodes. The printer package however contains all the logic needed to print the FSTNodes back to source code.

Hooking Mechanism

As in Section 5.2.4, a method of describing hooking points was described. This hooking method was also implemented for FeatureHouse. To do this, we created an additional node type in the XML tree, called FSTHook. A FSTHook is a nonterminal node, which is used to contain all the individual nodes and subtrees that require composition in a specific place. If when parsing an XML file the begin comment is found, we set the properties of the FSTHook to contain the identifier of the XML node this hook is associated with and whether it will be applied *before*, ore *after* that XML node. All XML nodes that are found between the begin and the end comments are then added as child nodes to the FSTHook.

Once we have FSTTree structure prepared, we need to make composition alterations to enable hooking. This extra functionality was added to the main composition method in *FSTGenComposer*. During normal composition, when two nonterminal are to be to composed, each of the compatible children are then recursively composed.

[1]https://github.com/deankramer/featurehouse

However, if the node being superimposed is a FSTHook, all the children nodes of that hook are then either added before or after the specific widget in the that base node.

For composition, nodes have to be checked for their compatibility. In FeatureHouse, this is carried out by checking the *type* and *name* of the FSTNode. In normal programming languages, this can normally be easy to map e.g. a method declaration and its signature. With XML nodes this is more difficult, as XML document do not always use the same node attributes. Therefore, during XML parsing, we need to make some assumptions on which attributes should be used to map to the FSTNode name. If a name is not added to each FSTNode, this can cause composition errors in XML documents that contain many instances of the same XMLNode type. In our implementation, we currently consider the following attributes to be appropriate names: `android:id`, `android:name`, `name`, `key`, and `id`. Nodes that do not have one of those attributes are likely to cause composition errors. In future work, we hope to design an approach to automatically finding appropriate XML attributes at runtime, to enable it work over more general XML documents.

7.2.3 Runtime Composition

In this dissertation, we do not propose a language solution to support static or runtime composition. We propose to use existing language solutions that exist already. As we are targeting the Android mobile platform, we need to use a language that have a Java basis. FOP language extensions for Java for example JAK (Batory et al., 2004), FeatureHouse (Apel, Kastner and Lengauer, 2009) currently only support compile time variability. Also, while there has been some work on other similar language languages e.g. DynamicDeltaJ (Damiani and Schaefer, 2011), there is yet to be an available implementation. For this reason, for runtime variability, we use a similar language extension, the Context-Oriented Programming language (COP), JCOP (Appeltauer et al., 2010).

In COP, runtime variability are implemented as *layers* (Hirschfeld et al., 2008). Each layer represents a specific runtime concern that can be activated or deactivated dynamically. These layers crosscut different source code modules with partial methods, which like FOP, refine methods to include additional executable logic. These layers can be *within* a class, whereby all adaptations for a given class are contained with that class. Alternately, in some COP languages e.g. JCOP, layers can be declared *outside* a class, whereby all adaptations for that particular concern are contained within their

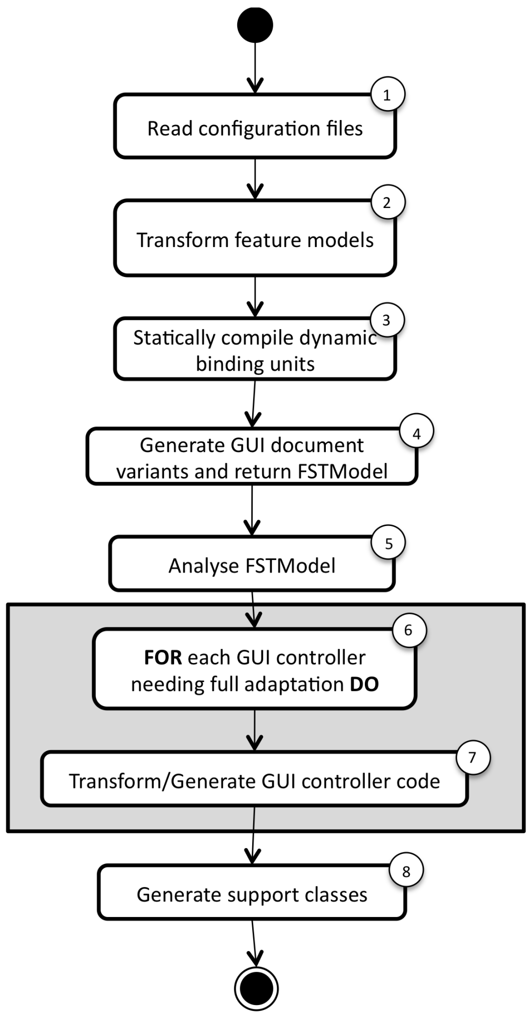

1
Read configuration files
2
Transform feature models
3
Statically compile dynamic binding units
4
Generate GUI document variants and return FSTModel
5
Analyse FSTModel
6
FOR each GUI controller needing full adaptation **DO**
7
Transform/Generate GUI controller code
8
Generate support classes

Figure 7.2: Source Generation & Transformations

own first class entity.

The similarity between FOP and COP means that automated source code transformation are feasible. However, for this work, we currently transform FOP code to JCOP manually. We accomplish this in a number of steps. First, we get the final compound source code refinements by statically composing all features within each dynamic binding unit. This is carried out using FeatureHouse (Apel, Kastner and Lengauer, 2009). Following this, we can check every dynamic binding unit for class refinements. If there is a class refinement in a dynamic binding unit, a layer for that binding unit can be added to the base version of that class. Following this, the different class refinement methods can be added to that layer. Finally, we need to alter calls to previous refinements of that method. In FeatureHouse these method calls are declared with the keyword `original`, followed with the method arguments, similar to the Java super call. In JCOP, to continue the execution to the following partial method definition, the keyword `proceed` is used. This means, we just need to swap the `original` keyword for `proceed`.

We have depicted this transformation in Listing 7.1, using the scenario application. In this example, we can see a base version of class *ContentDetails*, a class refinement in the dynamic binding unit, and the final transformed class. In the transformed class, there is a nested layer for the feature containing that refinement, named `NoUserReview`. This layer contains the method refinement, with the `original` keyword changed to `proceed`.

Listing 7.1: FOP-COP Transformation

```
//Base Class
public class ContentDetails extends Activity {
  ...
  public void onCreate_contentreviews(ViewGroup vg) {
        ....
  }
  ...
}

//Class Refinement in feature ‘‘NoUserReview"
public class EditEventActivity extends Activity {
  ...
  public void onCreate_contentreviews(ViewGroup vg) {
    original();
    btnSaveReview = (Button) vg.findViewById(R.id.btnSaveReview);
    btnSaveReview.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View arg0) {
        Toast toast = Toast.makeText(mContext, R.string.error_cantsendreview,
                                               Toast.LENGTH_LONG);
        }
      }
  }
}

//Transformed JCOP Class
public class EditEventActivity extends Activity {
  public void onCreate_contentreviews(ViewGroup vg) {...}
  public layer NoUserReview {
    public void onCreate_contentreviews(ViewGroup vg) {
      proceed();
      btnSaveReview = (Button) vg.findViewById(R.id.btnSaveReview);
      btnSaveReview.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View arg0) {
        Toast toast = Toast.makeText(mContext, R.string.error_cantsendreview,
                                               Toast.LENGTH_LONG);
         }
       }
    }
    ...
  }
  ...
}
```

Special consideration has to be taken to ensure method refinements are executed in the correct order. This is due to feature compositions being generally not commutative (Apel et al., 2010). As a consequence of this, when combining static with dynamic composition, method refinements can be applied in the wrong order. To solve this, we create hook methods as proposed by (Rosenmüller et al., 2011a).

7.2.4 Source Generation & Transformation

For runtime adaptation, many components of the system need to be generated, outlined in Figure 7.2. While these have been described previously in some detail, here we present more implementation detail. We first start by reading the dynamic binding unit (DBU) and runtime GUI adaptation configuration files (1). The first describes the different DBUs or compound features to be featured in the product, each listing the different SPL features that will be contained within each DBU. The second configuration file is a list of different GUI classes e.g. Android Activities and Fragments, that may require the GUI to adapt once it is visible to the user. We then have to transform and refactor the feature model using the steps presented in Chapter 5 (2). Next in (3), we statically compose source code required for each DBU in the configuration. We then carry out GUI document variant generation (4). This produces all the different GUI document variants required to enable runtime adaptation.

Next, we need analyse the complete program, including source code, and GUI documents. In FeatureIDE similar to FeatureHouse, a FSTModel of the complete SPL can be created which we use for analysis in the source generation and transformation steps. We next get the complete structure of the project from FeatureHouse in the form of a simple tree of FSTNonTerminal and FSTTerminal nodes, called a FSTModel (5). We parse this to a model we can more easily manipulate, and transform. Next, we can firstly carry out the code transformations on the GUI classes.

GUI Controller Transformations

To adapt the GUI once it is visible to the user, a method to handle the adaptation is required. We know what GUI controllers require this type of adaptation from the configuration files parsed earlier. For each GUI controller in the configuration file (6), we need to carryout the appropriate source transformations (7). Each of these methods is copied over from a source code template file. The first is that a `reconfigure_` method is required for each GUI document that GUI controller handles, with the template shown in Listing 7.2. Each instance of this method handles the adaptation for parts of the screen handled by a particular GUI document. We therefore need to change each instance to include the name of the GUI document in the method signature name, and update the elements surrounded in $$ with the appropriate information using the FSTModel including the root widget id.

Listing 7.2: Reconfigure_ method template

```
public void reconfigure_() {
  ViewGroup root = (ViewGroup) this.findViewById(android.R.id.content);
  ArrayList<View> instances = findAllDocumentInstances(root, $$DOCUMENTWIDGETREF$$);
  int cases = instances.size();
  ArrayList<Set<Integer>> alteredWidgetsArray = new ArrayList<Set<Integer>>();
  for (int i = 0; i<cases;i++) {
    alteredWidgetsArray.add(new HashSet<Integer>(alteredWidgets));
  }

  for (int i = 0; i<cases;i++) {
    localalteredWidgets = alteredWidgetsArray.get(i);
    ViewGroup instance = (ViewGroup) instances.get(i);
    ViewGroup newUI = (ViewGroup) this.getLayoutInflater().inflate(
      dsplrg.getResourceVariant($$DOCUMENTFULLNAME$$), null);
    onCreate_$$DOCUMENTNAME$$(newUI);
    if (widgetIndexes.isEmpty()) {
      getPositionIndexes(newUI);
    }
    for (Integer widget : localalteredWidgets) {
      moveWidget(widget.intValue(), newUI, instance, null);
    }
    newUI.removeAllViews();
  }
  instances.clear();
  alteredWidgetsArray.clear();
  localalteredWidgets.clear();
  widgetIndexes.clear();
}
```

Next in (8), we need to generate the support classes, as described next.

Different support classes are generated as part of the source transformation and generation process. The generation of classes is outlined in Figure 7.3. First, we need to generate the state transfer class (1). This class is responsible for transferring widget state between the widget instances being replaced. Based on the analysed FSTModel, we can establish which types of widgets will be getting replaced at runtime. Based on these different types, we generate the state transfer class by copying the required GUI state retention templates, described in Section 5.2.6 into the state transfer class.

Next, for every DBU in the product configuration (2), we need to generate a Feature class (3). Each Feature class contains the methods described in Section 5.3.3. We use the `addSourceRefinements` and `removeSourceRefinements` methods to call the JCOP specific keywords for activating specific layers. The other responsibility is to return to the GUI Manager the list of widgets that need adapting because of that dynamic binding unit. We can find all widgets that require adapting by checking each GUI XML file within each feature that is part of a dynamic binding unit. As we described in Chapter 5, in a GUI document refinement, when a refinement is added, all parent nodes need to be present in the tree. These parent nodes should only have their widget identifiers in the node attributes. Because of this, we can scan each XML to see what nodes have more than one attribute. Therefore if a node is found to have more than a single attribute, we can assume it is either a new node, or has a new or altered attributes. We then get the identifier for this node, and add it to a list. When all GUI documents have been checked, we can add the list to the `getAllAlteredWidgets` method.

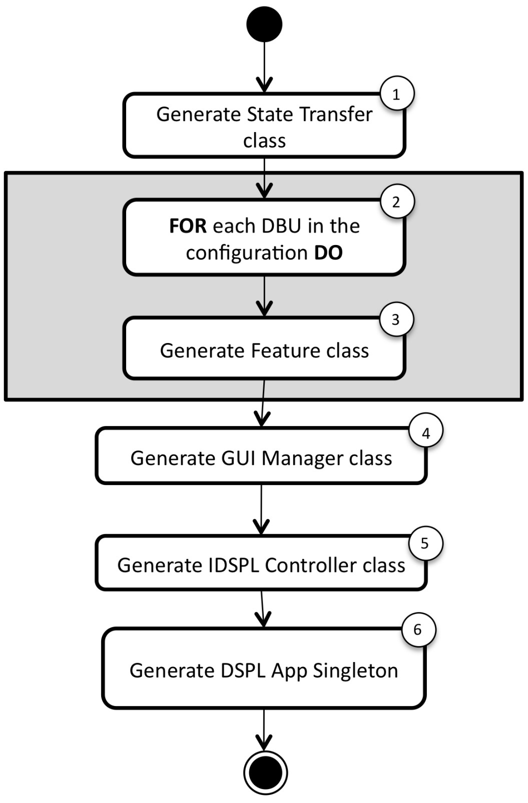

1
Generate State Transfer class
2
FOR each DBU in the configuration **DO**
3
Generate Feature class
4
Generate GUI Manager class
5
Generate IDSPL Controller class
6
Generate DSPL App Singleton

Figure 7.3: Generated Support Classes

Generated Classes

The last three steps include generating the GUI manager (4), the IDSPL controller interface class (5), and DSPLApp singleton. These classes are static files that require no further changes to them. The GUI manager is a unspecialised class for the project that receives the configuration updates from FeatureDroid. The IDSPL controller interface is an interface that all Android classes e.g. Fragments and Activities that require adaptation need to implement.

Variant Manager

The variant manager is created to manage the GUI document variants, by way of returning the correct variant when requested. The manager is generated as a single class and contains a single public method used by rest of the application to request the correct variant for a specific GUI document. In Android, GUI document references are stored as static integers. Therefore, to invoke the correct GUi document retrieval method, this can be accomplished with a switch, as depicted in Listing 7.3.

Listing 7.3: GUI Document Variant Retrieval Method

```java
public int getResourceVariant(int resourceName) {
  int resource;
  switch(resourceName) {
    case R.layout.mainscreen: resource = get_mainscreen();
                                        break;
    ...
    default: resource = resourceName;
                                 break;
  }
  return resource;
}
```

7.3 FeatureDroid

This section describes the DSPL middleware named FeatureDroid, a middleware for handling runtime context acquisition, and DSPL configuration management. This middleware was not developed completely from scratch, but uses various components from FeatureIDE, including feature model and configuration classes. The system can be used both as a single central entity as its own application, or as part of another application as library, described in Chapter 6. This system can be broken down into two systems, ContextEngine, and DSPL Management. First we describe ContextEngine.

7.3.1 ContextEngine

The ContextEngine is a general purpose context acquisition engine for context-aware applications (Kramer et al., 2011)[2]. It has been developed as a self contained system that can be used for other context-aware applications. ContextEngine can be used as a library by a single application, or can be used as a separate centralised context management system.

ContextEngine is made up of three main components including a context manager, a database, and a set of atomic context components. The main context manager is designed to handle the lifecycle of each component, and be an interface point by applications and the DSPL management component. The manager can be interfaced in two ways:

- **Object Calls:** When ContextEngine is packaged with an application, calls to the engine can be made directly with standard Java object method calls.
- **AIDL Services:** Context-Engine also publishes its services through two interfaces declared using Android Interface Definition Language (AIDL) as shown in Listing 7.4. This then allows applications in different sandboxes to communicate with the manager. The first interface is designed for context requests and tasks not requiring a response. This includes tasks e.g. requesting new contexts to be acquired, or deploying new contexts. The other is for requests requiring instant response for e.g. the context value of a specific context.

Listing 7.4: ContextEngine Interfaces

```
//Interface 1: Context Requests
interface IContextsDefinition {
  void setupContexts(String path);
  boolean addContextValues(in String appKey, in String componentName, in String[] contextValues);
  boolean addContextValue(in String appKey, in String componentName, in String contextValue);
  void addSpecificContextValue(in String appKey, in String componentName, in String contextValue, in
       String numericData1, in String numericData2);
  void addRange(in String appKey, in String componentName, in String minValue, in String maxValue, in
       String contextValue);
  boolean newComposite(in String appKey, in String compositeName);
  boolean addToComposite(in String appKey, in String componentName, in String compositeName);
  void addRule(in String componentName, in String[] condition, in String result);
  boolean startComposite(in String compositeName);
  void copyDexFile(in String appKey, in String newDex, in String[] contexts, in String packageName, int
       permission);
```

[2] Source code available at: https://github.com/deankramer/ContextEngine

```
}
//Interface 2: Synchronous Requests
interface ISynchronousCommunication {
  List<String> getContextList();
  String getContextValue(in String componentName);
  boolean isComponentDeployed(in String appkey, in String component);
}
```

The context manager manages the lifecycle of all context components in the system. This is carried out by only having contexts that have been requested by applications active in the system. Applications make requests for contexts by adding new context value definitions to a context component. If the component is not currently active, it is then started. When application no long requires a context, it can remove its context-value definitions. Context components that have no context-value definitions are then shutdown. This ensures that contexts only execute while they are required.

Context Types

Depending on the context being monitored, different types of context components will be needed. Particularly with the Android platform, there are multiple sources for context data, that may need to be acquired differently. For this we have a set of four different context component types to suit these different requirements:

- **ListenerComponent**. This component type captures raw sensory data. Sensor data is received by the component from a sensor event. Examples of sensors that are used with this component include light level, ambient temperature, gravity, proximity, gyroscope, and magnetic field[3]. To receive sensory data, this component type implements the Android *SensorEventListener* class interface, receiving sensor updates using `onSensorChanged` method. This method is called by the Android *SensorManager*, and passes a float array of values. Depending on the sensor, this value can relate to different sensor values e.g. light lumens in the light sensor.

- **MonitorComponent**. This component type captures data that is not considered sensory. These context types are instead broadcasted globally, which can be listened for. Examples of the type of contexts this component can be used for include device battery level and WIFI status. To capture these events, we use *BroadcastReceiver* objects to listen for specific system broadcasts, also known

[3]If the hardware support also exists

as intents. Each context component registers for a particular intent action e.g. `Intent.ACTION_BATTERY_CHANGED` for battery changes.

- **PreferenceChangeComponent**. This component type captures data from application preferences. The implementation shares similarities to the Listener-Component, whereby it implements a listener interface. In this case, the *OnSharedPreferenceChangeListener* is implemented, with updates being received using the `onSharedPreferenceChanged` method. Preferences are essentially key-values, each having a specific name or identifier, and a value which can be a number of different types.

- **LocationContext**. This component type captures location data from the location services e.g. GPS/GLONASS. This component provides an abstraction of different location data, and provides means to compute if the device is nearby particular locations.

Each of these context component types extend an abstract base class named *ContextComponent*. This class provides the basic data structures and logic required by contexts. To use composite contexts, we created a component called *CompositeComponent*. This context component functions similarly to the *MonitorComponent*, in which it listens for global context broadcasts. The differences lie in that it holds the context values for more than a single context, and it also runs rules over the values to deduce higher level context information.

Context Broadcasts

Context changes are broadcasted asynchronously, to each application, and composite context listening for changes. This is handled using Intent Broadcasts. Intent Broadcasts allow for IPC to multiple applications, whereby interested applications *listen* for these broadcasts using Broadcast Receivers. Within a broadcast, several pieces of information are added including the intent action, the name of the context, the date and time of the context change, and the new context value itself. Intent actions act a method of filtering all system messages that are broadcasted e.g. battery status. These broadcasts are sent and received using the `sendNotification` and `setupMonitor` methods as depicted in Listing 7.5.

Listing 7.5: SendNotification and setupMonitor Implementations

```
//The listening intent needed for the intentfilter
```

```java
public static final String CONTEXT_INTENT = "uk.ac.mdse.contextengine.CONTEXT_CHANGE";

public void sendNotification() {
  Intent intent = new Intent();
  intent.setAction(CONTEXT_INTENT);
  intent.putExtra(CONTEXT_NAME, name);
  intent.putExtra(CONTEXT_DATE, Calendar.getInstance().toString());
  intent.putExtra(CONTEXT_INFORMATION, contextInformation);
  sendBroadcast(intent);
  }

IntentFilter intFilter = new IntentFilter(CONTEXT_INTENT);
private void setupMonitor() {
  contextMonitor = new BroadcastReceiver() {

    @Override
    public void onReceive(Context c, Intent intent) {
      String contextname = intent.getExtras().getString(CONTEXT_NAME);
      String contextValue = intent.getExtras().getBoolean(CONTEXT_INFORMATION);
      ...
      }
    };

    context.registerReceiver(contextMonitor, intFilter);
  }
```

Runtime Component Deployment

In Android, application source code is compiled into a single .dex file. To enable the ability to copy context components from one application to the middleware, the application needs to be compiled into multiple .dex files.

This can be carried out by using a customised Ant build script, which can be found in the appendix of this thesis. Within the script, we extend the target name ”-dex” that contains the different tasks related to source compilation and its conversion to .dex. In this target, we need to separate the code for the deployable contexts from the rest of the application. For this reason, it is suitable to have all the contexts within their own java package, as this can then be more easily separated. We can then compile each directory separately into its own dex file. After the compilation has taken place, the .dex file containing the context components is then packaged within a jar file. Finally, the jar file is added the applications assets folder, so it is then included as a movable asset in the application.

When the application starts up, the application needs to check if the contexts have been deployed previously. This can be carried out either by storing a value in the application storage, or the application can query the management system to see if the context(s) exist. The application can query the management system through the use of the `isComponentDeployed` method in the ContextEngine interfaces.

Once deployed, context components are loaded from .dex files dynamically using dynamic class loaders (Liang and Bracha, 1998). In the ContextEngine, necessary information needed to load component e.g the package name, and what dex file contains the class are stored in the ContextEngine database. Multiple components can be contained within a single .dex file, but are loaded individually based on the requirements of the system. These files are copied over to the SD card, and the copied to the private application area of the ContextEngine when components are deployed at runtime.

7.3.2 DSPL Management

The main DSPL manager handles the management of DSPL applications, by supporting feature model, and configuration management. As mentioned earlier, this manager was created using altered components from FeatureIDE.

The main DSPL management has been implemented to function as either a *Service* when in the case of an shared middleware, or as a single set of classes, when part of a single application. Service APIs are defined similar to ContextEngine using AIDL interfaces as shown in Listing 7.6. This interface allows new DSPL instances to be started using the `setupDSPL` method. For this a unique application identifier, and a path is required. This path should point to a specific folder in the device filesystem e.g. the SD card, containing both the feature model and the initial configuration file. If FeatureDroid is packaged with the DSPL program, both files are taken from the Android application assets folder. The user can also change the DSPL configuration at runtime by either setting a configuration file using the `setConfiguration` method, or by changing the selection state of individual features using the `changeFeatureSelection` method. The return values of the methods correspond to the successful execution of themselves. Particularly when attempting to change the selection state of different features, the program can only be reconfigured if the resulting configuration is valid.

Listing 7.6: DSPL Service Interface

```
interface IDSPLManager {
        void addFeatureModel(String applicationid, String path);
        void setupDSPL(String applicationid, String path);
        List<String> getActiveFeatures(String applicationid);
```

```
    boolean setConfiguration(String applicationid, String path, String config);
    boolean changeFeatureSelection(String featureName, boolean state, boolean force);
}
```

Notifying external applications of configuration changes is handled using Intent Broadcasts, similarly to ContextEngine. In these broadcasts, the application identifier, and all features that are now active are broadcasted.

Interfacing with ContextEngine

As we described earlier, ContextEngine is a separate component for acquiring and managing context information. When context information changes, the feature configurations of each DSPL need to be updated. In our current implementation, we package ContextEngine as an internal component of FeatureDroid. We therefore do not need to use intent broadcasts, which are more computationally expensive, and instead can use object calls in the same application sandbox. This is carried out by creating a new class which extends main *ContextEngineCore* manager class. In this extended class, we override the method `setupContextMonitor`, responsible for receiving context events and forwarding them, which now instead updates the feature configuration for each application.

7.3.3 Deploying FeatureDroid

FeatureDroid can be used in two scenarios: as part of a DSPL application, or as a separate entity. To use FeatureDroid as part of a DSPL application, the developer needs to include it in the compilation of their application. To do this, the developer needs to compile the FeatureDroid source code as an Android library project. This can then be included into the main application using two methods. The first is that the developer manually copies the compiled FeatureDroid.jar archive into the application's lib folder, and link it in the Java builder. The second approach is to add the FeatureDroid application as an Android library dependency. Then when the main application is being compiled, FeatureDroid will also be compiled and added during the build process.

When FeatureDroid is used as a separate entity, FeatureDroid needs to be deployed as any other application on Android. First, the FeatureDroid application needs to be compiled, this time not as a library but as a standard application. During the compilation, this application can be then signed, and uploaded to a mobile app store including the Google Play Store. If this is not wanted, that compiled app can then be installed to devices directly from their development machine, or via a website by sharing the application installer .apk file. To install this application outside of the Google Play Store, the developer/user will need to ensure they have altered their system settings to allow this.

7.4 Summary

In this chapter, we presented the implementation details of the different prototypes created for both phases of the DSPL, the design stage, and runtime phase. For the design phase, feature modelling and composition tools were implemented to assist in the development of a DSPL with runtime GUI support. For the runtime phase, the middleware FeatureDroid for handling the configuration of each DSPL application is presented. This is includes the ContextEngine for context acquisition and management. These prototypes were developed as open source software, and are available to download.

8 Evaluation

Contents

8.1 Introduction

In this thesis so far, an approach to enable unified GUI adaptation in DSPLs has been proposed, with details of developed tools and prototypes given. In this chapter, focus is given on a final evaluation of the proposed approach, using the prototypes presented in the last chapter. The goal is to validate that the approach actually meets the goals of this thesis, and to evaluate the extent to which it scales. To validate the approach, we use two different scenario DSPLs. We further carry out different scalability experiments to help assess the feasibility of our approach over different cases.

This chapter can be broken down into the following: Section 8.2 describes our validation using scenario DSPLs, with details of the variability categories tested. Next in Section 8.3, we describe the scalability experiments and present our results. Final discussion based on our validation and scalability experiments is given in Section 8.4. This chapter is then summarised in Section 8.5.

8.2 Scenario-Based Evaluation

The first half of this evaluation is scenario-based. A full and complete evaluation of our approach will require extensive testing. In Chapter 4, we described and discussed 9 different categories of variability in the GUI, and different runtime adaptation requirements that should be met by the approach. Therefore, to fully evaluate our approach would require testing each permutation of these variability categories and requirements. Furthermore, this will need to be carried out over many different types of GUIs, with different GUI structures, and on screen widgets. To reduce the evaluation effort, we carry out a number of scenario based examples for our validation. Using these different scenarios, we attempt to ensure a good coverage of validation testing by combining different variability categories, and adaptation requirements. We use the following subjects as description and discussion areas:

- **System and Context Models**. Due to the size of the system and context feature models, we will describe the different application and context features added to these models, including the context rules. The full models can be found in the Appendix.
- **GUI adaptation using Feature-Oriented Programming**. We illustrate and describe how we implement the scenario variability using both FOP for GUI documents, and source code.
- **Runtime GUI adaptation**. We describe how the scenario will behave at runtime, and where full runtime adaptation is required, describe and explain the additional auxiliary configuration, and source code required to enable it.

This evaluation while concentrating on validating the proposed approach, can also allow us to understand more qualitative issues. The first of these includes understanding the strength and weaknesses of the approach. Secondly, we also discuss the issue of GUI adaptation unification, and what insights we have learnt from carrying out this evaluation. This, including the scalability tests later on in the Chapter, should provide us with an appropriate level of evaluation and final discussion. Next, we describe the different SPLs we use for our scenarios.

8.2.1 Scenario SPLs

To evaluate our approach we use two different scenario SPLs for mobile devices. In this evaluation, we use two different SPLs that differ in size and complexity. Particularly, one of the SPLs is a made of synthetic GUI screens, and the other is a real world application. Due to these vastly different SPLs, we can claim that our approach can function in both synthetic and real world applications.

The SPLs used are the following:

- **ContentStore**. A content store application is a small application based on a few different scenario GUI screens. This store can distribute different content e.g. applications, video, and music, and can support different age groups. The type of content the store can distribute depends on the location of the device, due to different possible distribution licenses. Content can be retrieved by the user either by downloading or by streaming. This scenario application is the one we introduced in Section 1.1.2, and have used throughout this thesis.

- **K-9 Email**. A popular opensource email client available for Android[1]. This application has over 110,000 lines of code, and was refactored into an DSPL with 33 features. This email program can be used for viewing, managing, and sending email. Particular dynamic features we focus on include how to deal with connectivity and storage limitations for downloading attachments, and gestures to support email deletion.

An important aspect to reiterate is the rationale behind the choice of these two scenario SPLs. These scenarios are not intended to display, or justify the scalability of our approach. Scalability of our approach is handled next in Section 8.3. These scenarios are used for validating our approach can handle the different types of GUI variability, which we outlined and discussed in Chapter 4.

Using these scenario SPLs, we can test the different GUI variability categories, as explained next.

8.2.2 Variability Categories

Back in Chapter 4, we introduced 9 different categories of GUI variability. These categories of variability are based on different aspects of the GUI. Many of these categories however are often implemented in the same way. As an example, if we consider

[1]https://code.google.com/p/k9mail/

the categories *properties of UI elements* and *visual appearance*, they are both implemented on Android by either adding or refining different XML node attributes. As a result, to test all categories of GUI variability, we do not need to test every category individually. Instead, we can test each of these GUI variability categories based on their method of implementation. In a GUI document, GUIs are described in a set of tree nodes, with each node having a set of properties. This means that we can generalise our GUI variability into the following categories of implementation variability:

- **GUI Elements.** This type of variability includes adding and removing whole GUI elements e.g. buttons, and text fields. These GUI elements can be singular, or can be containers of other GUI elements. This implementation variability can be used for implementing presentation units, UI element, and layout variability described in Chapter 4.
- **GUI Element Properties.** This type of variability encompasses alterations to different properties of GUI elements. When considering GUI documents, this means altering what attributes exist, and or, the attribute values. This implementation variability can be used for UI element property, dialogue, and visual appearance variability in Chapter 4.
- **GUI Behaviour.** This type of variability includes different arbitrary logic that also needs to be applied on reconfiguration. This logic can be used for defining behaviour e.g. gesture handlers, and also alter UI element event handlers. For the sake of our evaluation, it can also encompass other visual adaptations that can only be carried out in source code logic.

We argue these different variability categories form a representative set of variability cases that this work is designed to unify. Based on this, we use these types of variability within the SPLs we have presented earlier. The first categories we consider are GUI elements.

GUI Elements

In this example, we show how we handle variability for GUI elements. These GUI elements can both either be a single widget e.g. buttons, and text fields, or can be layout widgets that act as containers of other GUI elements. For this type of variability, we use the ContentStore application home screen. On this screen, depending on the

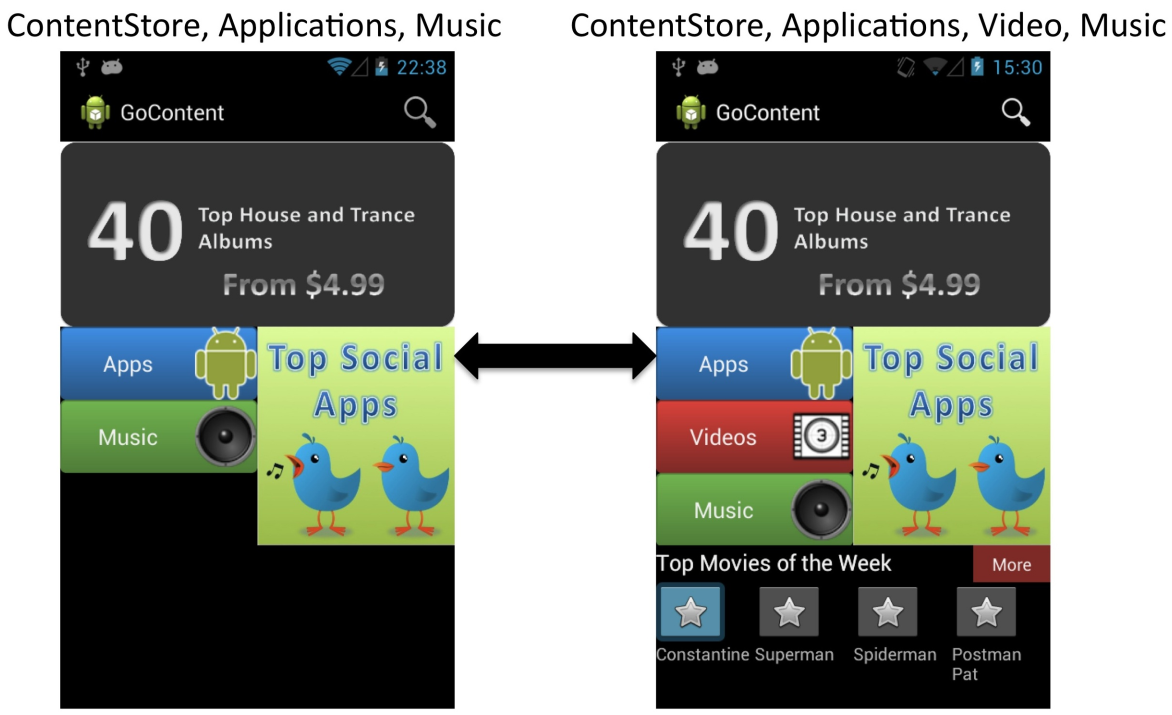


Figure 8.1: Screenshots of the main screen

geographical location of the device, the user has access to different types of content including applications, videos, and music. This screen therefore has different elements related to different content types. In Figure 8.1, we illustrate screenshots of the screen with and without the videos feature.

Firstly we consider the context and application feature models. For this scenario, we need the following contexts:

- **GeographicRegion**: This context checks the coarse gained location of the mobile device, specifically just to which continent. This context has the context values of north america, south america, europe, africa, asia, and oceania. These different regions are checked using a range of different longitudes and latitudes. This context will deduce which types of content are available to the user.

- **WifiContext**: This context monitors the state of the Wifi connection on the device. This can be either an off state (WifiOff), a disconnected state (WifiNotConnected), or connected state (WifiConnected).

- **TelephonyContext**: This context monitors the state of the telephony connection on the device. The phone can be connected in either a 2G connection (Tele2G), 3G connection (Tele3G), or 4G/LTE connection (Tele4G).

- **Internet**: This context is a composite context made up of the Wifi and telephony contexts to deduce if the device has capable internet connectivity. If either Tele3G, Tele4G, or WifiConnected contexts are active, the device can be understood to have internet connectivity (InternetOn). This context is required for the user to be able to browse the different content available in the store.

In the application feature model, we need the features *ContentStore, Applications, Music,* and *Videos*. The ContentStore feature in terms of the home screen has the base structure of the GUI including advertisements and GUI containers. Buttons and screen areas for specific content types are contained within *Applications, Music,* and *Videos*. Since the Applications feature is default in all regions, we do not need a context rule to enable it, and instead have it active in the initial configuration. Music and Videos however are supported in different geographic regions. Music is enabled in North and South America, Europe, and Oceania. This is enabled with the following context rule:

$$North\_america \vee South\_america \vee Europe \vee Oceania \Rightarrow Music$$

Videos can only be downloaded in North America, and Europe. This is handled in the following context rule:

$$North\_america \vee Europe \Rightarrow Videos$$

Listing 8.1: Activity_main_screen Document Refinement

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/framelayout" >
    <LinearLayout android:id="@+id/mainlayout" >
        <LinearLayout android:id="@+id/corebuttons" >
            <LinearLayout android:id="@+id/contenttypes" >
            <!-- @start after android:id’=’’@+id/btnApps’’ -->
                <Button
                    android:id="@+id/videos"
                    android:background="@drawable/btnVideos"
                    android:text="@string/videos"
                    ..../>
            <!-- @end after android:id’=’’@+id/btnApps’’ -->
            </LinearLayout>
        </LinearLayout>
        <LinearLayout
```

```
            android:id="@+id/videoads"
            android:orientation="vertical"
            ....>
            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content" >
                <TextView
                    android:id="@+id/videoAdvertTitle"
                    android:text="@string/advert_title"
                    ..../>
                <Button
                    android:id="@+id/btnVideoMore"
                    android:onClick="movieAdClick"
                    android:text="@string/More"
                    ..../>
            </LinearLayout>
            <LinearLayout
                android:id="@+id/videoadcontainer"
                ..../>
        </LinearLayout>
    </LinearLayout>
</FrameLayout>
```

Next, lets look at the different source refinements required. The videos feature contains a number of refinements to the home screen. Firstly, there needs to be a button to take the user to the section of the application destined for selling video content. As we want to be sure that the video button is always directly after the button for apps, we add the `after` refinement ordering statement. The second is a group of adverts to show the most popular videos to the user.

In Listing 8.1, we have the GUI document refinement to add the Videos button, and the container for holding video advertisements in the LinearLayout named `videoads`. This container contains a title, a button to take the user to a larger list of videos in order of popularity, and an empty container named `videoadcontainer`. This container will be used to hold each individual advertisement, each of which are instances of a GUI document named `videoadview`.

Next we describe the supporting class refinements for this example.

In Listing 8.2, we show the refinement added to the `MainScreen` activity. As described in Chapter 5, we need to have an `onCreate` method to handle all initialisation operations for the `activity_main_screen` GUI document. Included in this method refinement are the operations to add a listener to handle touch events on the main video button, and logic to populate the `videoadcontainer` element container with instances of the `videoadview` GUI document for each movie.

Listing 8.2: MainScreen class Refinement

```java
public class MainScreen extends Activity {
  public void onCreate_activity_main_screen(ViewGroup vg) {
  original();
  Button btnVideo = (Button)vg.findViewById(R.id.btnVideo);
  btnVideo.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        gotoVideoStoreScreen();
      }
    });
  ArrayList<VideoAdvert> videoads = getVideoAdvertisements();
  ViewGroup root = (ViewGroup) this.findViewById(R.id.videoadcontainer);
  for (VideoAdvert ad : videoads) {
    LinearLayout newMovie = (LinearLayout)View.inflate(this, R.layout.videoadview, null);
    TextView movieName = (TextView) newMovie.findViewById(R.id.videoName);
    movieName.setText(ad.getName());
    ImageView movieImg = (ImageView) newMovie.findViewById(R.id.videoImage);
    movieImg.setImageBitmap(ad.getImage());
    root.addView(newMovie);
   }
}
```

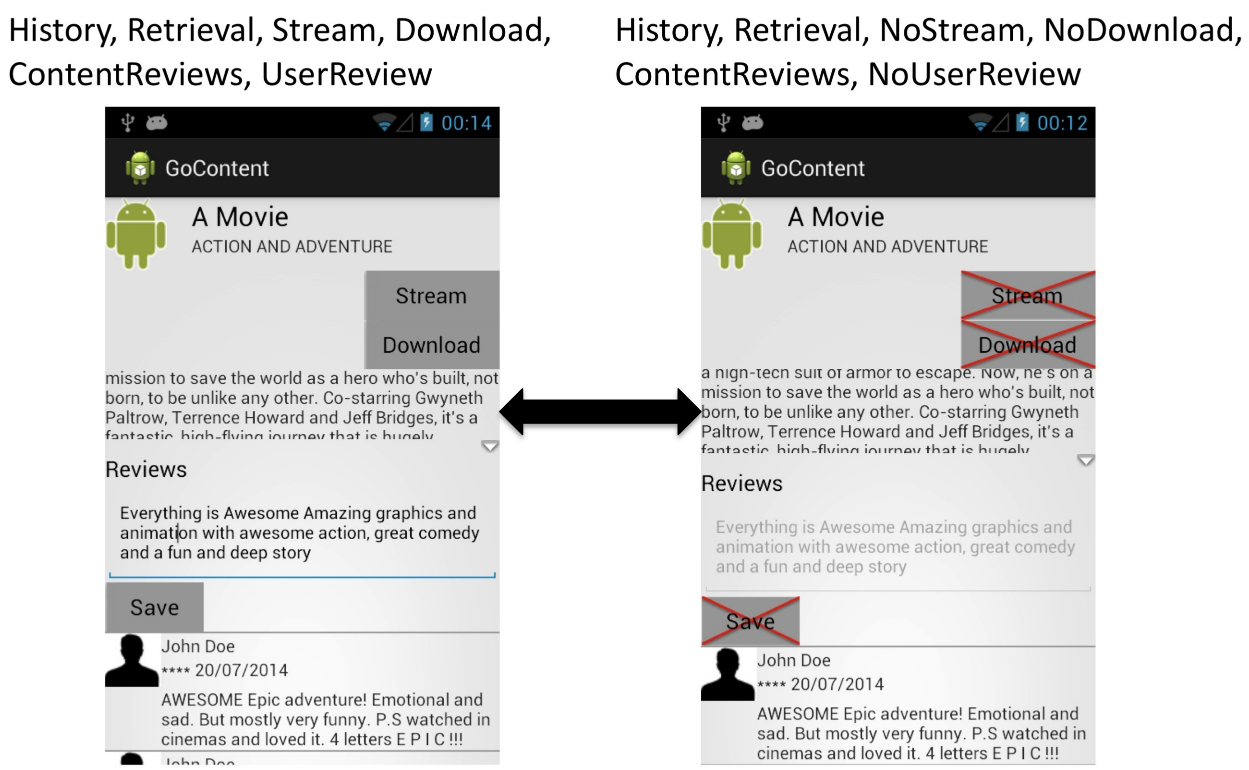
Lastly, we need to consider the adaptation timing of this variability. We have chosen to apply this adaptation, only when the screen is created, for two reasons. There firstly is a large amount of visual change to the GUI which could confuse the user if adapted. Secondly, it is very unlikely that the user will change geographic continents during the use of this app. This therefore means that we do not need to add the activity to the runtime adaptation configuration file.

Next, we describe an example of how we implement variability in GUI element properties.

GUI Element Properties

This next scenario considers how we adapt the properties of GUI elements on the screen. For this scenario we consider the screen designed to allow the user to download, install (in the case of applications), stream (in the case of videos), and review that particular content. This screen is affected by the network connectivity and remaining battery of the device. For this scenario, we use the same context model as used in the previous scenario except with different context rules for the application. To download or review content, the device must have a connection that is either 3G, LTE, or WiFi, with an internet connection. In Figure 8.2, we illustrate two different configurations of the GUI, with each of the relevant features stated above.

First, we define the following contexts in our context model:


History, Retrieval, Stream, Download, ContentReviews, UserReview
History, Retrieval, NoStream, NoDownload, ContentReviews, NoUserReview
00:14
GoContent
Stream
Download
Save
00:12
GoContent
Stream
Download
Save

A Movie
ACTION AND ADVENTURE

mission to save the world as a hero who's built, not born, to be unlike any other. Co-starring Gwyneth Paltrow, Terrence Howard and Jeff Bridges, it's a fantastic high-flying journey that is hugely

Reviews

Everything is Awesome Amazing graphics and animation with awesome action, great comedy and a fun and deep story

John Doe
**** 20/07/2014
AWESOME Epic adventure! Emotional and sad. But mostly very funny. P.S watched in cinemas and loved it. 4 letters E P I C !!!

John Doe

A Movie
ACTION AND ADVENTURE

a high-tech suit of armor to escape. Now, he's on a mission to save the world as a hero who's built, not born, to be unlike any other. Co-starring Gwyneth Paltrow, Terrence Howard and Jeff Bridges, it's a fantastic high-flying journey that is hugely

Reviews

Everything is Awesome Amazing graphics and animation with awesome action, great comedy and a fun and deep story

John Doe
**** 20/07/2014
AWESOME Epic adventure! Emotional and sad. But mostly very funny. P.S watched in cinemas and loved it. 4 letters E P I C !!!

Figure 8.2: Screenshots of the main screen

- **Battery**: This context monitors the amount of remaining device battery charge. This can be either very high (BatteryHigh), medium (BatteryMedium), or low (BatteryLow).

- **WifiContext**: This context monitors the state of the Wifi connection on the device. This can be either an off state (WifiOff), a disconnected state (WifiNotConnected), or connected state (WifiConnected).

- **TelephonyContext**: This context monitors the state of the telephony connection on the device. The phone can be connected in either a 2G connection (Tele2G), 3G connection (Tele3G), or 4G/LTE connection (Tele4G).

- **ExternalStorageSpaceContext**: This context monitors the amount of storage capacity in the main external storage area of the device, normally the SD card. The device can either have high amount (StorageHigh), medium amount (StorageMed), or low amount (StorageLow) of remaining capacity.

- **Internet**: This context is a composite context made up of the Wifi and telephony contexts to deduce if the device has capable internet connectivity. If either

Tele3G, Tele4G, or WifiConnected contexts are active, the device can be understood to have internet connectivity (InternetOn).

- **DataSync**: This context is a composite context made up of the Internet and battery to deduce when is the right time to carry out connections to the main context store.

These contexts will drive the main adaptation in this GUI. The main contexts that we plan to affect the GUI are ExternalStorageSpaceContext, and DataSync. DataSync makes the user unable to download or stream content. It also stops the user from being able to write and save a review of the content. These are expressed with the following rule:

$$DataSyncOff \Rightarrow NoUserReview \wedge NoDownload \wedge NoStream$$

StorageLow on the other hand only results in the user being unable to download the content due to lack of space. The user can still review the content and stream it. This is expressed in the following:

$$StorageLow \Rightarrow NoDownload$$

Next, we need to implement the different GUI document refinements.

Listing 8.3: GUI Document Refinements

```
//contentdetailheader.xml refinement in feature ‘‘Download’’
<LinearLayout android:id="@+id/contentdetailheader">
  <Button android:id="@+id/downcloudcontent"
            android:layout_width"wrap_content"
            android:layout_height="wrap_content"
            android:background="@drawable/can"
            android:text="@string/download" />
</LinearLayout>
//contentdetailheader.xml refinement in feature ‘‘NoDownload’’
<LinearLayout android:id="@+id/contentdetailheader">
  <Button android:id="@+id/downcloudcontent"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:background="@drawable/cant"
            android:text="@string/download" />
</LinearLayout>
//contentreviews.xml refinement in feature ‘‘NoUserReview’’
<LinearLayout android:id="@+id/contentreviews" >
  <EditText android:id="@+id/txtreviewsValue"
             android:enabled="false" />
  <Button android:id="@+id/btnSaveReview"
             android:background="@drawable/cant" />
```

```
</LinearLayout>
```

In Listing 8.3, we have excerpts of different GUI document refinements. We do not have room to have all refinements, so we illustrate just some of them. As this GUI is broken down over multiple GUI documents, we needs to refine multiple GUI documents. Two of the refinements are for the `contentdetailheader.xml` document, which both add a download button, but with different properties. The first is designed for when the Download feature is activated, and the user can download that specific content. The second is when the NoDownload feature is activated, and the user cannot download content due to specific contexts. The last refinement disables a button responsible for saving user reviews for a specific item. The next set of refinements required for this scenario include refinements to the Android activity sourcecode.

We consider how the GUI should be behave in Figure 8.4. This excerpt implements the behaviour required by the system if the application cannot connect to the content store because of a lack of internet connectivity. This sourcecode refinement is used along with the with the visual changes made to `contentreviews.xml`. This refinement implements an error message to the user by use of an Android Toast. To implement this behaviour we had to add additional logic to the GUI document initialisation method `onCreate_contentreviews`.

Listing 8.4: GUI Document Initialisation Refinement for ContentDetails.java class in Feature NoUserReview

```java
public class ContentDetails extends Activity {
  public void onCreate_contentreviews(ViewGroup vg) {
    original();
    btnSaveReview = (Button) vg.findViewById(R.id.btnSaveReview);
    btnSaveReview.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View arg0) {
        Toast toast = Toast.makeText(mContext, R.string.error_cantsendreview,
                                          Toast.LENGTH_LONG);
      }
  }
```

Next, because we plan to disable the textfields when the device has no connection to the internet, we need to be sure the state from the review textfield is retained during adaptation. This is done by creating a state retention file for the class `EditText`. Finally, as this adaptation is needed when the GUI is active, and not just when it is created, we need to add the activity class to the runtime adaptation configuration file. This is carried simply by adding the name of the class file to the text based configuration file.

Listing 8.5: Auxiliary files needed for scenario

```
//State retention for the Text fields in the widget templates.
<NEWWIDGET>.setText(<OLDWIDGET>.getText());

//Activity added to the runtime adaptation configuration file
ContentDetails.java
```

GUI Behaviour

For this type of variability, we consider a gesture for remove items in a list. This example is used in K9 Mail, and allows the user to easily remove emails by swiping the item in the list. Variability in this scenario is not driven by device contexts like the other scenarios, but driven by properties of the application, which are alterable by the user. This adaptation does not require any variability in a GUI document, only in the source code. The more simplistic version of this gesture given in Chapter 5 would require a base version of this gesture to nullify the gesture. The reason for this is that changes made in the source are not automatically reversed, and therefore default values must be set. In this example however, because gesture support is not handled using the same Android APIs, and the appropriate method is called based on the direction of swipe, we can adapt the gestures using standard FOP method refinements.

In the standard downloadable version of K9 Mail, gestures are used for selecting/deselecting messages in the mail list GUI (MessageListFragment). To implement this, we make the base version of the two methods for handling swiping events for the fragment empty. We then add refinements to the features ”leftToRight” and ”rightToLeft” to handle the operations to select the correct email message from the list, and delete it. In Listings 8.6, showing the base version of the two `onSwipe` methods, and the leftToRight method refinement.

Listing 8.6: The “Swipe to Delete” Gesture

```
//Base version (MessageListFragment.java)
public void onSwipeRightToLeft(final MotionEvent e1, final MotionEvent e2){
}
public void onSwipeLeftToRight(final MotionEvent e1, final MotionEvent e2){
}

//Feature leftToRight (MessageListFragment.java)
public void onSwipeLeftToRight(final MotionEvent e1, final MotionEvent e2){
	int x = (int) downMotion.getRawX();
	int y = (int) downMotion.getRawY();
	Rect headerRect = new Rect();
	mListView.getGlobalVisibleRect(headerRect);
```

```
        if (headerRect.contains(x, y)) {
            int[] listPosition = new int[2];
            mListView.getLocationOnScreen(listPosition);
            int listX = x - listPosition[0];
            int listY = y - listPosition[1];
            int listViewPosition = mListView.pointToPosition(listX, listY);
            onDelete(getMessage(listViewPosition));
        }}
```

As this adaptation does not require any GUI document refinements, we do not need to add the `MessageListFragment` class to the full runtime adaptation configuration file.

8.2.3 Summary of Scenarios

In this section, we presented different types of variability using two different scenario DSPLs. Based on the different implementation variability categories, we believe our approach can be used for implementing each of the variability types presented in Chapter 4. Our observations based on carrying out these cases will be discussed later in the Chapter, along with the scalability results. In the next section, we attempt to understand more how the approach will scale, which will help us assess the feasibility of the approach.

8.3 Scalability Evaluation

Because our approach to handling dynamic variability in GUI documents requires the compile time generation of each possibly used variant, scalability tests were carried out following the scenario SPLs. The goal of these tests are to evaluate how feasible our approach is in different situations. To do this, we are interested how our approach behaves with different levels of variability and GUI document sizes. This can be broken down into the following areas:

- **Generation Time**: This is the time the tool takes to generate the needed GUI document variants and the source code needed to support runtime adaptation. For this metric, we do not consider the time required for configuration file reading, and automatic feature model refactoring.
- **Application Size**: This metric considers how the size of the application will change with the increased GUI document size and variability. This metric will

be measured at two specific points. First is the size of the compiled Android installation file, or the `.apk` file produced following compilation, measured on the development machine. The second is the size of the application once installed on the Android device. This is measured by checking the ”Apps” section of the Android Settings application.

- **Adaptation Time**: This final metric measures the time it takes for a complete adaptation cycle to take place. This time assumes a new configuration, and therefore does not consider the time the DSPL middleware takes to deduce a new configuration. To measure the time of a configuration, we start the application with only the base feature active, and then change the configuration to the configuration with all features active. We then do the reconfiguration again, reversing from all features active to just the base feature. We repeat this 1000 times to fetch the average time for these adaptations.

In these scalability tests, we do not test the scalability of non GUI adaptation including normal method extensions. Our approach as explained in Section 7.2.3 uses the Context-Oriented Programming language JCOP (Appeltauer et al., 2010) to handle runtime Java composition. In benchmark tests carried out on JCOP (Schuster et al., 2011), it was found their approach was 16% slower than standard conditional if branches.

To test these different areas, we constructed a basic SPL for handling variability of a single GUI document. This SPL contains a single mandatory feature, and 20 optional features.

The mandatory feature contains the base version of the GUI document. In our experiment, one of the independent variables are different GUI document sizes. We test different randomly created GUI documents that are <1KB, 2KB, 4KB, and 8KB. The <1KB contains just a container layout widget needed for putting the widgets inside. Inside the 2KB GUI document contains a number of different widgets including labels, textfields, and images. To create the 4KB and 8KB documents, we multiplied the widgets of the 2KB document as appropriate.

Each optional feature contains a refinement for the `activity_main` GUI document. Feature relationships can greatly influence the feasibility of our approach by altering the total number of variants generated. By choosing all additional features to be optional, we can judge our approach closer with variability worse case scenarios. Each refinement within a feature adds an additional button to the base GUI document, depicted in Listing 8.7. This widget type was chosen as it is a highly common widget used in GUIs. Because of this selection, we can not guarantee that the same results will be obtained from other widgets including textfields, checkboxes, and other custom widgets. We expect that deviation from our results will depend on the number of properties the developer uses with that widget, and the complexity required to use it at runtime including rendering and interaction.

Listing 8.7: GUI Refinement for Feature 1

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/outercontainer">
    <LinearLayout android:id="@+id/container">
      <Button
        android:id="@+id/button1"
        android:layout_width="41dp"
        android:layout_height="wrap_content"
        android:text="1" />

    </LinearLayout>
</LinearLayout>
```

Finally, to measure adaptation time, we created a basic Android application. For each test, we copy the variants and support code generated by our tools to the Android application. Then, we measure the time to adapt the GUI as explained earlier in this section.

These experiments are carried out using the following equipment:

- Lenovo Thinkpad T440p laptop, with an Intel i5 Processor, 8GB RAM, standard hard disk drive, and Windows 8.1
- Samsung Nexus S phone, with Android 4.2.1.

8.3.1 Results

In this subsection, we presented the results of our scalability tests. While originally the decision was made to have 20 optional features, because we considered the generation time as approaching unfeasibility at 14 features, we decided to end experimentation at this number of features. For all results data, please see Appendix C.

Generation Time

In Figure 8.3, we depict evaluation results for the generation time of our approach in logarithmic scale. The generation time varied between 1.11 seconds for a <1KB GUI

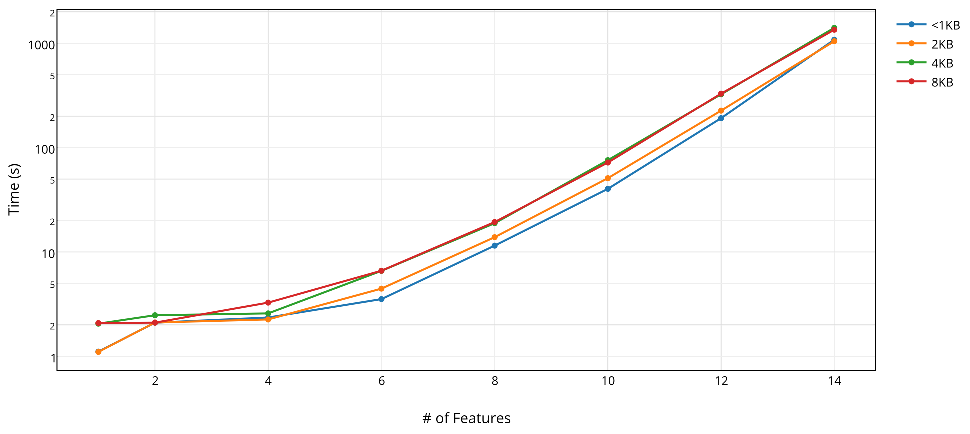


Figure 8.3: Generation Time (log)

Document with 1 feature to 1405.89 seconds (23.43mins) for a 4KB GUI Document with 14 optional features. The rate of increase is closely shared between each of GUI Document sizes.

We observed while running this experiment that file IO contributes to a considerable amount for the tool to generate all variants. Because our tool uses FeatureHouse, each of the variants generated are placed in their own configuration folder. Our tool therefore not only renames the variant file name to contain the variant identifier, but it also moves the file into a single collective folder for all variants. During generation, we found this operation to take a considerable amount of time. This experiment therefore we believe is highly dependant on the type of storage used. As our experiment was carried out on a standard laptop hard drive, we would expect a considerable time difference if a solid state drive was used.

Application Size

In Figures 8.4 & 8.5, we show the increase in application installation size, and installation .apk binary size in logarithmic scale. The application size increase varied from 2KB for a <1KB GUI document with 1 feature to 16538KB for a 8KB GUI document with 14 optional features. The installation binary size increase also varied from 0.01MB for a <1KB GUI document with 1 feature to 19.07MB for a 8KB GUI document with 14 optional features. This shows that the rate of increase is far higher for application size, than the rate of increase for the generation. We understand that the difference in size for the application installation and installation binary is due to (de)compression of the different application resources during compilation and installation. This group of results appears to rise far faster than the other two result categories. However, because the absolute size of all the variants is in the regions of megabytes, it is still less of an issue than the generation time result.

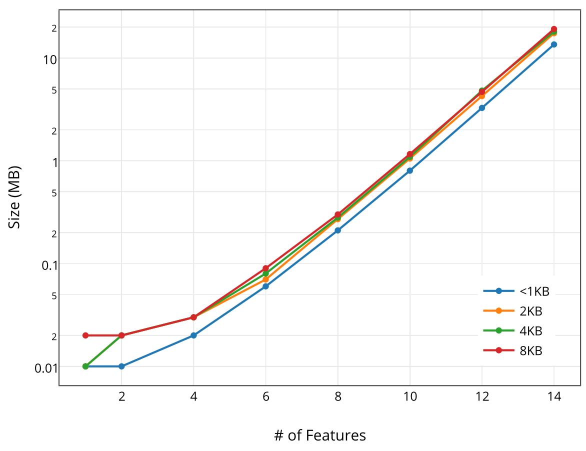


Figure 8.4: Installation Size Inc. (log)

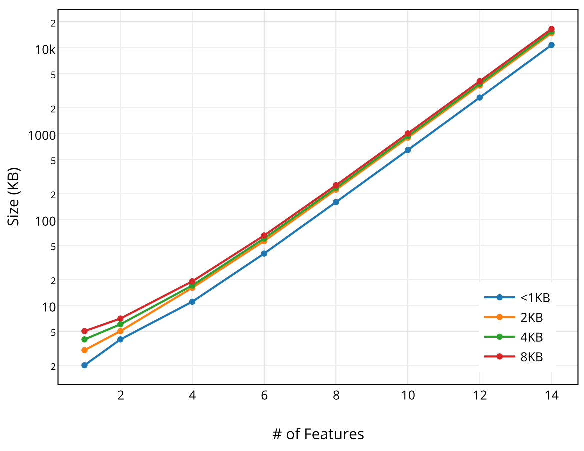


Figure 8.5: .apk File Size Inc. (log)

Adaptation Time

Finally, the results of the adaptation time are presented in Figure 8.6. Adaptation time scales far better than the generation time, and application size. Time to adapt the GUI was between 2.44ms for a GUI Document $<$1KB with a single feature, and 63.72ms for a GUI Document of 8KB with 14 optional features. This therefore means that a GUI tree can be adapted between 16 and 409 times a second. This means that in many cases, the adaptation process can be carried out faster than the screen can refresh, at 60Hz. Unsurprisingly, the time to adapt increased both with GUI Document size, and with the number of optional features.

We can see that the size of the GUI Document plays a role in the performance when comparing the time of each of the different sizes at the same number of features. This difference is most likely due to the time it takes for Android to inflate a new GUI subtree instance from each GUI Document variant. As the GUI Document size increases, more widget instances are needed in each of the GUI subtrees, which will subsequently take longer.

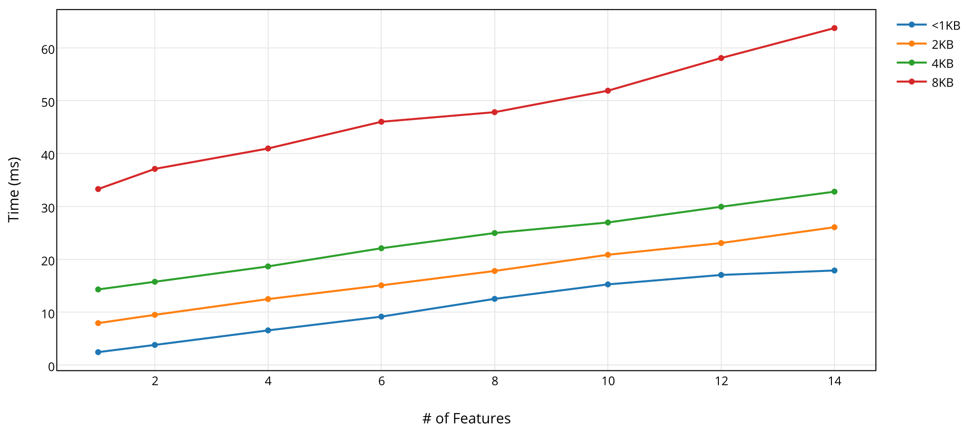


Figure 8.6: Adaptation Time

8.4 Discussion

Based on the experimentation given in the previous sections, we discuss our evaluation findings. To discuss our findings, we consider the following areas of interest: *unification, scalability,* and *extensibility*. We also compare our approach with previous approaches, which act as a baseline for our approach.

8.4.1 Unification

The core challenges described in Chapter 3 relate specifically to the unification of GUI adaptation, and context as a feature. By using our approach, we can enable the ability to apply GUI refinements both at design time at runtime. During our scenario SPLs, we discovered that in some cases we could not always disable features to disable specific elements of the GUI. In DSPLs that consider only logic, when particular functionality is not wanted anymore, that feature can be unbound, therefore removing that functionality. When considering the GUI, this was not always the same. In the example of the content download/review screen of the content store DSPL, in a static product, the feature to allow for downloading views may not be selected. When implementing this variability, this would mean that the download button is not included. If however the feature is deactivated in a DSPL product because of a lack of internet connection, this will cause the button to disappear which could possibly confuse the user. To ensure usability and user trust, we believe it would be more suitable to disable the button or alter the functionality of the button to inform the user of the situation. In these cases, we need to have additional features for this runtime adaptation, for example we introduced a “NoDownload” feature in the content store application. Depending on the kind of adaptation required, the developer will need to decide whether the runtime adaptation should result the same as a static product.

When handling context as a feature using feature models we find both advantages and disadvantages. Firstly, with our proposed DSPL middleware, using feature models, we have the benefit of being able to treat context as a reconfigurable entity of the system. This means that when contexts are not wanted or needed further, the system can cease its use. One disadvantage is that this requires finite models and that means more context deduction has to be carried out in the context components instead of the context model. This is because every value that we want to include in a context rule must relate to a feature name in one of the feature models. In instances of checking against many different values, for ease, it is better to aggregate different values into single context values, to avoid the feature models becoming too large and complex. An example of this includes handling country by country content available in the content store application. Instead of having a propositional rule for each country, have the context component aggregate different countries into the same context value when they have a shared fate.

8.4.2 Scalability

In our scalability tests, we have seen that our approach is broadly feasible in the situations of 8 to 12 dynamic binding units of GUI document variability. The cost of GUI unification using our approach can be seen to be quite high, and some might suggest to place GUI adaptation only in source code and not GUI documents. This approach however more scalable, will increase code scattering. While conventionally, code scattering can degrade a program’s comprehensibility, we believe this can be magnified when GUI code is scattered across different artefacts and languages. In our prototype platform, there is not always a simplistic corresponding class *getter* or *setter* for every GUI document property, complicating the issue of managing a single concern over many artefacts and languages. Also, with GUI documents, platform IDEs often allow the developer to preview and implement the GUI using drag and drop visual editors. If GUI adaptation is moved to source code, this could result in the developer no longer being able to develop the GUI using these tools.

A large issue with our approach relates to the amount of repetitive code generated with each GUI document variant. Repetitive source can become more of an issue in larger GUI trees, as the whole tree is present in every variant. For this we suggest to decompose the GUI document. By decomposing the GUI document, we essentially break the GUI document down in to multiple parts, that can be loaded separately and used at runtime. Depending on the location of the GUI document variability, highly variable sections of the document can be separated from the main GUI document, therefore lowering the amount of repetitive code. This not only helps lower the amount of storage required for an application installation, but can also make the runtime adaptation more efficient. At runtime, every time there is an adaptation, a new variant of that GUI document is reloaded before adding, removing, or swapping the appropriate widgets. As presented in the scalability tests, the time to adapt is dependant on the size of the GUI document. Consequently, the smaller the GUI document, the faster that GUI document instance will adapt.

Overall, we can see that scalability can be a real issue with our approach. Later in our future, we discuss further work we wish to carry out to try and improve this.

8.4.3 Extensibility

Extensibility is an important property of any SPL as product requirements constantly change, allowing for the derivation of new products. Extensibility in a DSPL can be related to both the *domain engineering*, and *application engineering* phases of the SPL. By having an extensible domain, the developer can easily add new variability to an SPL at design time.

Extensions to the domain can be handled easily by further FOP refinements for source code, or GUI documents. If more widgets are required for runtime adaptation that require their state to be retained, a state retention template can be created for each widget. This can be for both Android system widgets, or widgets implemented in the application. Additionally, contexts can be developed for the DSPL. As we explained in the earlier chapters of this thesis, each context in the context feature model has a corresponding context component implementation. We can therefore extend the context-awareness of the domain by adding additional contexts to the context model, and develop the new context components to be used by that DSPL.

An extensible application on the other hand refers to the ability to add and modify the variability of the application. Application derivation in DSPLs can happen both before deployment, and after at runtime. Application extensions can be carried out by adding additional features and contexts after deviation but before compilation. Currently runtime extensibility is limited to contexts. This is due to the use of the Context-Oriented Programming language JCop. JCop is built on top of AspectJ, an Aspect-Oriented language. Runtime aspect weaving is currently not supported in the Dalvik virtual machine, used by Android. This makes it impossible to extend the actual application currently. While dynamic binding and class loading can be handled by Android, we currently know of no Android compatible languages to handle this similar to FOP languages, for example FeatureC++. Because the Android UI framework requires GUI documents to be statically referenced in source code, this means no additional GUI document variants can be added at runtime. Contexts however can be added after deployment to the management system using the API we provided.

8.4.4 Previous Approaches

In this work, we propose an approach to unifying GUI adaptation using, and extending DSPLs. It is important we look back at previous work briefly to compare our approach with those already proposed. It is obviously difficult to compare solid scalability data we gathered with previous approaches fairly, as we have to consider the differences in system languages, platforms, and form factors.

Different DSPL approaches have been proposed previously. These different DSPLs differ on a number of aspects, including language, platform, and support for dynamic and static variability. Many DSPL approaches rely on service-oriented architecture for implementation (Lee and Kang, 2006; Marinho et al., 2010; Parra et al., 2009; Lee et al., 2012). While there are differences between each of these approaches, in essence they all use services for logic implementation. During runtime, these different services can be binded together supporting runtime adaptation. Other pure language approaches include FeatureC++ (Rosenmüller et al., 2011b). FeatureC++ makes use of C++ language extensions and dynamic binding to achieve adaptation. Adaptation is implemented in Feature-Oriented Programming, which is then transformed into the declarator pattern. This approach also proposed the use of static binding with dynamic to achieve greater scalability and performance. The one aspect that these DSPLs all share is their lack of real GUI support.

These different DSPLs do not consider how to adapt the GUI explicitly. In many of these approaches, where the GUI is programmed in the same language as the program logic, it is possible to adapt the GUIs when they are created/instantiated. However, these approaches do not give any ability to handle GUI adaptations once the GUI has been created, and is visible to the user. This therefore leaves a static GUI once it has been displayed to the user. It could be possible to combine DSPLs with other existing adaptive GUI approaches, but that leaves large drawbacks. The first is that static and dynamic GUI adaptation is separately developed. In the cases where a given GUI refinement is required in both static and dynamic final products, the developer will need to design, implement, and maintain two different versions depending on if the adaptation is required statically, or dynamically. The second drawback is how it is managed in the SPL. To handle both static and dynamic versions of that GUI, the developer will need to use one of two approaches. The first would require many additional features to be added to the feature model, each of which would contain each version of the different GUIs. The second approach would be require adaptation to be added to the product after it has been derived. These two approaches add additional product complexity in terms of their configuration management. Using our approach, only GUI adaptation needs only to be designed, and implemented once. This alleviates the need for managing features that require specific binding times in a single feature model, as all features can be bound statically, or dynamically.

In addition, as the lack of existing DSPL support also stretches to the ability to handle more than a single language. DSPLs in literature traditionally handle a single language which is used to implement the whole system. GUI development however in modern platforms often requires the need for multiple languages through their use of document-oriented GUIs, or GUI description languages. Using current DSPL approaches will effectively mean the developer cannot use GUI description languages as there is no support for them. Composing these documents at runtime while possible, would in practice either require GUI framework alterations, or a new GUI framework designed to handle the GUI documents. The primary reason for this is during application composition on different platforms including iOS, Android, and Microsoft Windows the GUI documents are compiled also into internal proprietary binary formats.

Because existing DSPL approaches only handle adaptations in program logic which is more easily composable at runtime, these approaches scale far better than our approach when including GUI variability. Having said this, scalability in our approach is mostly affected by GUI variability. Program logic variability in DSPL should scale broadly inline with existing approaches as our approach is extension of existing languages.

8.5 Summary

In this chapter, an evaluation of the proposed work has been presented. It has been shown that our approach has good coverage in terms of the types of GUI variability, as specified in Chapter 4, that can be handled using approach. Coverage is of variability cases is handled through the use of scenario DSPLs, with different types of GUI implementation variability. While our approach does not scale as well as the other existing DSPL approaches, we believe it is feasible for the vast majority of cases. More data will be needed across real industrial case studies however to assist in the generalisation of these findings.

Part IV

Conclusion

9 Conclusions and Future Work

Contents

9.1 Introduction

In this thesis, an approach for unifying GUI adaptation and context modelling in DSPLs has been presented. This final chapter concludes this thesis providing an overall summary in Section 9.2. In Section 9.3, we summarise the main contributions of this work, and finally discuss future work to be carried out in Section 9.4.

9.2 Thesis Summary

In the high proliferation of smart devices, and mobile applications, people are interacting with computers in new and exciting ways. With this use of pervasive and ubiquitous computing, user requirements are always changing. Using context-aware adaptive computing, software can adapt to suit the user in different situations and help anticipate their actions. SPLs and FOSD provide methods, processes, and techniques to manage this need for requirement variability. This systematic consideration for variability helps provide strategies for artefact reuse across product families using annotative and compositional approaches. Traditional SPLs however, consider variability only statically, and therefore cannot adapt to user requirements at runtime.

In this thesis, we contribute to the vision of unification in DSPLs, allowing for software to be adaptable statically, and dynamically. Our work primarily concentrated on bringing design time and runtime GUI adaptation unification to DSPLs, allowing the developers to apply refinements to the GUI statically and dynamically. This work is designed to solve this challenge particularly for Document-Oriented GUIs, which are an increasingly common approach to implementing GUIs through the use of markup, and GUI description languages. For our approach, we have extended the use of FOP, a language paradigm for implementing SPLs, to the domain of GUIs. Using FOP, GUI variability can be implemented through the use of GUI document refinements, which refine and adapt that GUI document in a specific way. Along with the ability for stepwise refinement of GUI documents, we enable the ability to precisely control the positioning of different GUI elements during composition. GUI document refinements however only constitute half of our design time contributions. Next, we extend FOP to enable the implementation of source code based GUI adaptations. This can be broken down into multiple parts. We firstly propose to place all GUI document initialisation logic, which is required to ensure GUI elements adapted at runtime are initialised appropriately. Secondly, we give the developer the ability to implement GUI adaptations based in source code, for example gesture support. These different adaptations can be refined like all other FOP methods enabling runtime stepwise refinement. This contribution is important to runtime GUI adaptation as its execution is always carried out on adaptation, which is not guaranteed using standard FOP.

To support runtime adaptation, we create several different components. The first component was designed to manage and orchestrate the reconfiguration of the application. The second component is used to manage the different GUI document variants, which can be requested by other classes in the system at runtime. The third and last component manages state retention by transferring widget state between variants at adaptation.

In addition, we focus on treating context as a unified feature in the DSPL. This allows the develop to model the context and system features in a single modelling notation. It also can enable the use of dynamic context-acquisition. Context-acquisition historically in DSPLs has been a static process, whereby context information is continuously collected regardless of the systems requirements. In our work, we focus on including context modelling in feature models to allow a single modelling notation to be used to model both context and the system variability. To complement this, we develop a DSPL management system that can dynamically activate/deactivate context sensing and deploy new context components. These two complimenting parts of this work are achieved by treating context as a feature of the system in a DSPL.

To validate our proposed work, we carried out a combination of scenarios and scalability tests. For scenario based tests, we categorised different implementation variability types, and carried out an example scenario for each. These variability categories included whole widget additions and removals, widget attribute changes, and adaptation of a gesture in the GUI. In the scalability evaluation, we examined how the approach scales with different amounts of variability, and complexity in the GUI. Overall, we found that the approach shows promise to be scalable to 8-12 optional features for each GUI document. We found that in our tests, it was the time it took to generate each of the GUI variants to be main hindrance in handling great amounts of variability.

Next, we describe the different contributions of this thesis.

9.3 Thesis Contributions

The core aim of this thesis was to bring unified GUI adaptation to DSPLs. In the process of this thesis, a number of contributions to the field of Software Product Lines have been achieved, including:

1. **An extended modelling approach to handle system variability and context:** To support the domain engineering process, we propose a unified modelling approach for handling system variability and context. By using extended feature models, we avoid the need for supplementary models required by previous approaches. The models are then used at runtime by the DSPL management system to not only adapt the DSPL application, but dynamically use different contexts.

2. **A Feature-Oriented approach to GUI adaptation:** An approach to unified GUI adaptation was proposed. This approach is based on Feature-Oriented Programming, and allows GUI variability to be implemented in refinements, along with the rest of the application. These refinements are composed together at compile-time using superimposition, and allows for precise GUI element placement using refinement hooks.

3. **A mechanism for runtime GUI adaptation:** In conjunction with the adaptation approach in the last contribution, we proposed a mechanism for dynamically adapting the GUI at runtime. Our proposal is an approach that only updates the widgets requiring adaptation, and ensures GUI state is retained during adaptation.

4. **A centralised DSPL management system:** To avoid the need for a DSPL management system for each DSPL on a mobile device, we proposed a system that can be centralised. This management system can be part of a single mobile app, or centralised to managed multiple DSPL applications. It also includes a context-acquisition engine, to handle context sensing and basic higher level context deduction. This component can also allow for context data to be shared over many mobile apps, and DSPL instances. As the context acquisition is part of the DSPL management system, we can also handle context activations and deactivations based on the DSPL feature model.

5. **Tool support and developed prototypes:** To validate and evaluate our contribution, we developed several deliverables including tool support, and mobile middleware. Tool support was built via an extension to the FeatureIDE plugins for Eclipse IDE to handle context modelling, and source code generation and transformations. It generates the different GUI document variants based on the variability of the DSPL, and transforms the source code to handle runtime GUI adaptation. The mobile middleware, FeatureDroid, includes the DSPL management system named FeatureDroid, developed for the Android mobile platform to handle DSPL configuration management. This middleware also includes the context management system, ContextEngine, for handling context sensing, and reconfiguration based on context.

One key impact of this work, in finding a solution to the research challenges in this thesis, a number of different artefacts were produced. In particular, extensions to FeatureHouse to enable composition have been included into the main project source code[1], enabling software developers to carry out static GUI document composition. It is also possible that in the future, the components developed to enable runtime GUI adaptation will be developed further, and be included as an additional plugin to FeatureIDE. We believe the interest by the research community to include our work

[1]The main FeatureHouse source code can be found at: https://github.com/joliebig/featurehouse

helps highlight its relevance to developers and researchers. In the next section, we describe and discuss areas in which further research is required.

9.4 Future work

While we argue this thesis provides a feasible answer to handling GUI adaptation in DSPLs, this we believe is just the beginning. In this section we aim to discuss some of the areas of future which we believe warrant further investigation, and development.

By developing our context models using feature models, we have been able to create a product where contexts can be activated or deactivated allowing for context acquisition to be self-aware to an extent. Currently, if the developer wants to alter the parameters of a context component e.g. the delay between getting a users location, the developer needs to have multiple similar context components. It would be interesting to extend our context models and DSPL management system to allow for context components to be adapted also allowing for the system to be fully self-aware and adaptive. Appropriate consideration will be needed to avoid multiple applications having conflicting context component parameters when context is acquired centrally.

Application extensibility was discussed to be a particular feature that is missing from our implemented solution. This is in part due to the need for GUI document variants, and also due to the use of the COP language, JCOP. To recap, we chose to follow the approach of generating all variants because runtime composition of the GUI documents using current platform APIs was not possible. This is not to say the idea of runtime composition is impossible on all platforms, just that we viewed the development effort to be too high. We also aimed to have an approach that we believe can be applicable across more than a single platform. It is possible that additional development could be carried out to parse the GUI documents that had not been preprocessed on some platforms. Particularly with Android If these two different components can be implemented, it would be interesting to conduct experiments to find the advantages and disadvantages of the different approaches.

In this dissertation, we have concentrated on compiled native mobile applications. With the rise and shift from native applications to web applications, this presents us with the interesting question of how this work can be adapted to the mobile web. We expect that while our approach to static composition could be adapted to work with HTML files, the real question is how the runtime GUI composition could be carried out. While work previously has used DSPLs with the web (Parra et al., 2009; Alferez and Pelechano, 2011), this has always been server-side. Modern HTML5 enables the ability to develop web applications that can be used offline, this means that server-side solutions cannot work offline. Solutions for runtime logic adaptation in JavaScript include ContextJS (Lincke et al., 2011). Just like our solution, it is possible a FOP extension for Javascript could be implemented for FeatureHouse, with similar FOP-COP source-code transformations for runtime adaptation. While it is possible an adapted version of our approach could be used with DOM elements in a web page, it should be possible to apply each refinement to the DOM tree without the need for whole variants, as used in this work. This therefore would be more scalable than the solution proposed for native applications.

Program validation, and verification are very important activities to ensure a program remains bug free, and meets the requirements it is designed for. This is no less true for GUIs in a SPL. GUI documents help enforce separation of concerns by forcing the developer to implement the view external from the data, and the business logic of the application in design patterns including the Model-View-Controller. This separation however, means there are multiple linked artefacts in the application. As these GUI documents are often only interpreted at runtime, they are often not statically checked at compile time. We therefore need to ensure that no inconsistencies between these artefacts occur. These inconsistencies can be two ways, between the controller and view, and equally between the view and the controller. Examples of these bugs include attempting to add event listeners to non-existent buttons in the GUI document, or referencing an onClick method in the GUI document that is not implemented in the controller using the document. While these inconsistencies can be easily managed in standard applications, SPLs can make it increasingly more difficult to check these artefact dependencies, and therefore these bugs could occur far more easily. To help find some of these inconsistencies, we suggest the following checks:

- **Controller-View**: This checks the consistency between view references made in the controller to the view. In this check, we attempt to ensure widgets referenced in controllers exist within the GUI document that is used.
- **View-Controller**: This checks the consistency between controller references made in the view to the controller. In this check, we attempt to ensure that, event handlers declared within the GUI documents are implemented within the controller.

As it is not always possible to understand the context of use for a GUI document in an application, we therefore suggest only potential warnings to the developer, not errors.

A Implementation specific scripts

Listing A.1: Ant build.xml target alterations to create context deployable jar

```
<!-- This is a modified version of the "dex-helper" macro. It added the "input-dir" and
        "output-dex-file" required attributes.
        Configurable macro, which allows to pass as parameters input directory,
        output directory, output dex filename and external libraries to dex (optional) -->
    <macrodef name="dex-helper-mod">
        <attribute name="input-dir" />
        <attribute name="output-dex-file" />
        <element name="external-libs" optional="yes" />
        <element name="extra-parameters" optional="yes" />
        <attribute name="nolocals" default="false" />
        <sequential>
            <!-- set the secondary dx input: the project (and library) jar files
                 If a pre-dex task sets it to something else this has no effect -->
            <if>
                <condition>
                    <isreference refid="out.dex.jar.input.ref" />
                </condition>
                <else>
                    <path id="out.dex.jar.input.ref">
                        <path refid="project.all.jars.path" />
                    </path>
                </else>
            </if>

            <echo>Converting compiled files and external libraries into @{output-dex-file}...</echo>
            <dex executable="${dx}"
                    output="@{output-dex-file}"
                    nolocals="@{nolocals}"
                    verbose="${verbose}">
                <path path="@{input-dir}"/>
                <path refid="out.dex.jar.input.ref" />
                <external-libs />
            </dex>
        </sequential>
    </macrodef>
```

```

<!-- This is a modified version of "-dex" target taken from $SDK/tools/ant/main_rules.xml -->
<!-- Converts this project's .class files into .dex files -->
<target name="-dex" depends="-compile, -post-compile, -obfuscate"
       unless="do.not.compile">
    <if condition="${manifest.hasCode}">
       <then>
          <!-- Create staging directories to store .class files to be converted to the -->
          <!-- default dex and the secondary dex. -->
          <mkdir dir="${out.classes.absolute.dir}.1"/>
          <mkdir dir="${out.classes.absolute.dir}.2"/>

          <!-- Primary dex to include everything but the concrete library implementation. -->
          <copy todir="${out.classes.absolute.dir}.1" >
             <fileset dir="${out.classes.absolute.dir}" >
                <exclude name="uk/ac/uwl/mdse/contextengine/components/**" />
             </fileset>
          </copy>
          <!-- Secondary dex to include the concrete library implementation. -->
          <copy todir="${out.classes.absolute.dir}.2" >
             <fileset dir="${out.classes.absolute.dir}" >
                <include name="uk/ac/uwl/mdse/contextengine/components/**" />
             </fileset>
          </copy>

          <!-- Compile .class files from the two stage directories to the apppropriate dex files. -->
          <dex-helper-mod input-dir="${out.classes.absolute.dir}.1"
             output-dex-file="${out.absolute.dir}/${dex.file.name}" />
          <mkdir dir="${out.absolute.dir}/secondary_dex_dir" />
          <dex-helper-mod input-dir="${out.classes.absolute.dir}.2"
             output-dex-file="${out.absolute.dir}/secondary_dex_dir/classes.dex" />
          <!-- Jar the secondary dex file so it can be consumed by the DexClassLoader. -->
          <!-- Package the output in the assets directory of the apk. -->
          <jar destfile="${asset.absolute.dir}/secondary_dex.jar"
              basedir="${out.absolute.dir}/secondary_dex_dir" includes="classes.dex" />
       </then>
       <else>
          <echo>hasCode = false. Skipping...</echo>
       </else>
    </if>
</target>
```

B Scenario Feature Models

Listing B.1: ContentStore Context Feature Model XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
        <featureModel chosenLayoutAlgorithm="1" showAllConstraints="true">
                <struct>
                        <and abstract="true" mandatory="true" name="Context">
                                <and abstract="true" mandatory="true" name="Device">
                                        <alt mandatory="true" name="Battery">
                                                <feature mandatory="true" name="BatteryLow">
                                                        <attribute domain="range" name="batlowdesc" value="0-10"
                                                             />
                                                </feature>
                                                <feature mandatory="true" name="BatteryMed">
                                                        <attribute domain="range" name="batmedesc" value="11-80"
                                                             />
                                                </feature>
                                                <feature mandatory="true" name="BatteryHigh">
                                                        <attribute domain="range" name="bathighdesc" value="
                                                             81-100"/>
                                                </feature>
                                        </alt>
                                        <alt name="Wifi">
                                                <feature mandatory="true" name="WifiOff"/>
                                                <feature mandatory="true" name="WifiNotConnected"/>
                                                <feature mandatory="true" name="WifiConnected"/>
                                        </alt>
                                        <alt name="Telephony">
                                                <feature mandatory="true" name="Tele2G"/>
                                                <feature mandatory="true" name="Tele3G"/>
                                                <feature mandatory="true" name="Tele4G"/>
                                        </alt>
                                </and>
                                <and abstract="true" name="AggContexts">
                                        <alt abstract="true" name="Internet">
                                                <feature mandatory="true" name="InternetOn"/>
                                                <feature mandatory="true" name="InternetOff"/>
                                        </alt>
```

```
                    <and abstract="true" name="DataSync">
                        <feature name="DataSyncOn"/>
                        <feature name="DataSyncOff"/>
                    </and>
                </and>
            </and>
        </struct>
        <constraints>
            <rule>
                <imp>
                    <disj>
                        <var>Tele3G</var>
                        <disj>
                            <var>Tele4G</var>
                            <var>WifiConnected</var>
                        </disj>
                    </disj>
                    <var>InternetOn</var>
                </imp>
            </rule>
            <rule>
                <imp>
                    <conj>
                        <disj>
                            <var>BatteryHigh</var>
                            <var>BatteryMed</var>
                        </disj>
                        <var>InternetOn</var>
                    </conj>
                    <var>DataSyncOn</var>
                </imp>
            </rule>
        </constraints>
        <comments/>
        <featureOrder userDefined="false"/>
    </featureModel>
```

C
Scalability Data

Doc. Size	# of Fea.	Variant Gen. Time (s)	Total App Size (KB)	Install. Size (MB)	Run. Adapt. Time (ms)
original	0	0	284	1.04	0
<1KB (246B)	1	1.11	286	1.05	2.448
-	2	2.101	288	1.05	3.811
-	4	2.356	295	1.06	6.561
-	6	3.535	324	1.10	9.163
-	8	11.503	443	1.25	12.524
-	10	40.274	927	1.84	15.259
-	12	191.518	2909	4.30	17.045
-	14	1082.182	11038	14.54	17.887
2KB	1	1.103	287	1.05	7.934
-	2	2.108	289	1.06	9.501
-	4	2.257	300	1.07	12.479
-	6	4.454	340	1.11	15.077
-	8	13.856	506	1.31	17.798
-	10	50.941	1179	2.09	20.86
-	12	226.134	3918	5.29	23.078
-	14	1046.036	15070	18.40	26.069

Table C.1: Scalability Data for <1KB and 2KB GUI Documents.

Doc. Size	# of Fea.	Variant Gen. Time (s)	Total App Size (KB)	Install. Size (MB)	Run. Adapt. Time (ms)
original	0	0	284	1.04	0
4KB	1	2.056	288	1.05	14.304
-	2	2.476	290	1.06	15.756
-	4	2.58	301	1.07	18.659
-	6	6.594	344	1.12	22.092
-	8	18.858	517	1.32	24.968
-	10	75.793	1220	2.13	26.986
-	12	324.176	4081	5.84	29.943
-	14	1405.889	15732	19.04	32.790
8KB	1	2.084	289	1.06	33.286
-	2	2.103	291	1.06	37.097
-	4	3.272	303	1.07	40.96
-	6	6.613	349	1.13	46.009
-	8	19.342	534	1.34	47.823
-	10	71.975	1288	2.20	51.881
-	12	328.904	4352	5.71	58.065
-	14	1348.312	16822	19.07	63.727

Table C.2: Scalability Data for 4KB and 8KB GUI Documents.

Bibliography

Abowd, G. D., Atkeson, C. G., Hong, J., Long, S., Kooper, R. and Pinkerton, M. (1997). Cyberguide: A mobile context-aware tour guide, *Wirel. Netw.* **3**(5): 421–433.

Acher, M. (2011). *Managing Multiple Feature Models: Foundataions, Language and Applications*, PhD thesis, Université De Nice-Sophia Antipolis.

Acher, M., Collet, P., Fleurey, F., Lahire, P., Moisan, S. and Rigault, J.-P. (2009). Modeling Context and Dynamic Adaptations with Feature Models, *4th International Workshop Models@run.time at Models 2009 (MRT'09)*, p. 10.

Alferez, G. and Pelechano, V. (2011). Context-aware autonomous web services in software product lines, *Software Product Line Conference (SPLC), 2011 15th International*, pp. 100 –109.

Alves, V., Matos Jr., P., Cole, L., Borba, P. and Ramalho, G. (2005). Extracting and evolving mobile games product lines, *in* H. Obbink and K. Pohl (eds), *Software Product Lines*, Vol. 3714 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 70–81.

Anfurrutia, F. I., Díaz, O. and Trujillo, S. (2007). On refining xml artifacts, *Proceedings of the 7th international conference on Web engineering*, ICWE'07, Springer-Verlag, Berlin, Heidelberg, pp. 473–478.

Apel, S. (2007). *The Role of Features and Aspects in Software Development*, PhD thesis, Otto-von-Guericke-University Magdeburg.

Apel, S., Janda, F., Trujillo, S. and Kästner, C. (2009). Model superimposition in software product lines, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, Springer-Verlag, Berlin, Heidelberg, pp. 4–19.

Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development, *Journal of Object Technology* **8**(5): 49–84.

Apel, S., Kastner, C. and Lengauer, C. (2009). Featurehouse: Language-independent, automated software composition, *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, IEEE Computer Society, Washington, DC, USA, pp. 221–231.

Apel, S., Leich, T., Rosenmüller, M. and Saake, G. (2005). Featurec++: on the symbiosis of feature-oriented and aspect-oriented programming, *Proceedings of the 4th international conference on Generative Programming and Component Engineering*, GPCE'05, Springer-Verlag, Berlin, Heidelberg, pp. 125–140.

Apel, S. and Lengauer, C. (2008). Superimposition: a language-independent approach to software composition, *Proceedings of the 7th international conference on Software composition*, SC'08, Springer-Verlag, Berlin, Heidelberg, pp. 20–35.

Apel, S., Lengauer, C., Möller, B. and Kästner, C. (2010). An algebraic foundation for automatic feature-based program synthesis, *Sci. Comput. Program.* **75**(11): 1022–1047.

Apel, S., Lengauer, C., Möller, B. and Krästner, C. (2008). An algebra for features and feature composition, *in* J. Meseguer and G. Rosu (eds), *Algebraic Methodology and Software Technology*, Vol. 5140 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 36–50.

Appeltauer, M., Hirschfeld, R. and Masuhara, H. (2009). Improving the development of context-dependent java applications with contextj, *International Workshop on Context-Oriented Programming*, COP '09, ACM, New York, NY, USA, pp. 5:1–5:5.

Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M. and Kawauchi, K. (2010). Event-specific Software Composition in Context-oriented Programming, *Proceedings of International Conference on Software Composition*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 50–65.

Avrahami, D. and Hudson, S. E. (2006). Responsiveness in instant messaging: Predictive models supporting inter-personal communication, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, ACM, New York, NY, USA, pp. 731–740.

Bass, L., Clements, P. and Kazman, R. (2003). *Software Architecture in Practice*, SEI Series in Software Engineering, Addison-Wesley.

Batory, D. (2005). Feature models, grammars, and propositional formulas, *in* H. Obbink and K. Pohl (eds), *Software Product Lines*, Vol. 3714 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 7–20.

Batory, D., Sarvela, J. and Rauschmayer, A. (2004). Scaling step-wise refinement, *IEEE Trans. Softw. Eng.* **30**: 355–371.

Behan, M. and Krejcar, O. (2012). Adaptive graphical user interface solution for modern user devices, *in* J.-S. Pan, S.-M. Chen and N. Nguyen (eds), *Intelligent Information and Database Systems*, Vol. 7197 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 411–420.

Benavides, D., Segura, S. and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review, *Inf. Syst.* **35**: 615–636.

Benavides, D., Trinidad, P. and Ruiz-Cortés, A. (2005). Automated reasoning on feature models, *Proceedings of the 17th international conference on Advanced Information Systems Engineering*, CAiSE'05, Springer-Verlag, Berlin, Heidelberg, pp. 491–503.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A. and Riboni, D. (2010). A survey of context modelling and reasoning techniques, *Pervasive and Mobile Computing* **6**(2): 161 – 180.

Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison-Wesley.

Boucher, Q., Abbasi, E., Hubaux, A., Perrouin, G., Acher, M. and Heymans, P. (2012). Towards more reliable configurators: A re-engineering perspective, *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, pp. 29 –32.

Brummermann, H., Keunecke, M. and Schmid, K. (2011). Variability issues in the evolution of information system ecosystems, *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, ACM, New York, NY, USA, pp. 159–164.

Bruntink, M., van Deursen, A., D'Hondt, M. and Tourwé, T. (2007). Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations, *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, ACM, New York, NY, USA, pp. 199–211.

Calvary, G., Coutaz, J. and Thevenin, D. (2001). A unifying reference framework for the development of plastic user interfaces, *in* M. Little and L. Nigay (eds), *Engineering for Human-Computer Interaction*, Vol. 2254 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 173–192.

Cetina, C., Giner, P., Fons, J. and Pelechano, V. (2009). Autonomic computing through reuse of variability models at runtime: The case of smart homes, *Computer* **42**(10): 37–43.

Chen, K., Zhang, W., Zhao, H. and Mei, H. (2005). An approach to constructing feature models based on requirements clustering, *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pp. 31 – 40.

Classen, A., Heymans, P. and Schobbens, P.-Y. (2008). What's in a feature: a requirements engineering perspective, *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 16–30.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*, 3rd edn, Addison-Wesley Professional.

Collignon, B., Vanderdonckt, J. and Calvary, G. (2008). Model-driven engineering of multi-target plastic user interfaces, *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, ICAS '08, IEEE Computer Society, Washington, DC, USA, pp. 7–14.

Conan, D., Rouvoy, R. and Seinturier, L. (2007). Scalable processing of context information with cosmos, *Proceedings of the 7th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'07, Springer-Verlag, Berlin, Heidelberg, pp. 210–224.

consortium, O. (n.d.). Frascati project, http://frascati.ow2.org.
URL: *http://frascati.ow2.org*

Cordy, M., Schobbens, P.-Y., Heymans, P. and Legay, A. (2013). Beyond boolean product-line model checking: dealing with feature attributes and multi-features, *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 472–481.

Coutaz, J., Balme, L., Alvaro, X., Calvary, G., Demeure, A. and Sottet, J.-S. (2007). An mde-soa approach to support plastic user interfaces in ambient spaces, *Proceedings of the 4th International Conference on Universal Access in Human-computer Interaction: Ambient Interaction*, UAHCI'07, Springer-Verlag, Berlin, Heidelberg, pp. 63–72.

Criado, J., Vicente-Chicote, C., Padilla, N. and Iribarne, L. (2010). A model-driven approach to graphical user interface runtime adaptation, *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems*, pp. 49–59.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

Czarnecki, K., Helsen, S. and Eisenecker, U. (2004). Staged configuration using feature models, *in* R. Nord (ed.), *Software Product Lines*, Vol. 3154 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 266–283.

Czarnecki, K., Helsen, S. and Eisenecker, U. W. (2005a). Formalizing cardinality-based feature models and their specialization, *Software Process: Improvement and Practice* **10**(1): 7–29.

Czarnecki, K., Helsen, S. and Eisenecker, U. W. (2005b). Staged configuration through specialization and multilevel configuration of feature models, *Software Process: Improvement and Practice* **10**(2): 143–169.

Czarnecki, K. and Wasowski, A. (2007). Feature diagrams and logics: There and back again, *the Proceedings of the 11th International Software Product Line Conference*, pp. 23–34.

Damiani, F. and Schaefer, I. (2011). Dynamic delta-oriented programming, *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, ACM, New York, NY, USA, pp. 34:1–34:8.

Daniele, L. M., Silva, E., Pires, L. F. and Sinderen, M. (2009). A soa-based platform-specific framework for context-aware mobile applications, *in* W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, C. Szyperski, R. Poler, M. Sinderen and R. Sanchis (eds), *Enterprise Interoperability*, Vol. 38 of *Lecture Notes in Business Information Processing*, Springer Berlin Heidelberg, pp. 25–37.

David, L., Endler, M., Barbosa, S. D. J. and Filho, J. V. (2011). Middleware support for context-aware mobile applications with adaptive multimodal user interfaces, *Proceedings of the 2011 Fourth International Conference on Ubi-Media Computing*, U-MEDIA '11, IEEE Computer Society, Washington, DC, USA, pp. 106–111.

Desmet, B., Vallejos, J., Costanza, P., De Meuter, W. and D'Hondt, T. (2007). Context-oriented domain analysis, *Proceedings of the 6th international and interdisciplinary conference on Modeling and using context*, CONTEXT'07, Springer-Verlag, Berlin, Heidelberg, pp. 178–191.

Dey, A. K. (2001). Understanding and using context, *Personal Ubiquitous Comput.* **5**: 4–7.

Dey, A. K. and Abowd, G. D. (2000). Cybreminder: A context-aware system for supporting reminders, *Proceedings of the 2Nd International Symposium on Handheld and Ubiquitous Computing*, HUC '00, Springer-Verlag, London, UK, UK, pp. 172–186.

Dey, A. K., Abowd, G. D. and Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, *Hum.-Comput. Interact.* **16**: 97–166.

Draheim, D., Lutteroth, C. and Weber, G. (2006). Graphical user interfaces as documents, *Proceedings of the 7th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: design centered HCI*, CHINZ '06, ACM, New York, NY, USA, pp. 67–74.

Feast, L. and Melles, G. (2010). *Epistemological Positions in Design Research: A Brief Review of the Literature*, number July, Connected, pp. 1–5.

Feigenspan, J., Kästner, C., Frisch, M., Dachselt, R. and Apel, S. (2010). Visual support for understanding product lines, *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, IEEE Computer Society, Washington, DC, USA, pp. 34–35.

Fernandes, P., Werner, C. and Teixeira, E. (2011). An approach for feature modeling of context-aware software product line, *Journal of Universal Computer Science* **17**(5): 807–829.

Findlater, L. and McGrenere, J. (2010). Beyond performance: Feature awareness in personalized interfaces, *Int. J. Hum.-Comput. Stud.* **68**(3): 121–137.

Gomaa, H. and Hashimoto, K. (2011). Dynamic software adaptation for service-oriented product lines, *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, ACM, New York, NY, USA, pp. 35:1–35:8.

Grolaux, D. (2007). *Transparent Migration and Adaptation in a Graphical User Interface Toolkit*, PhD thesis, Université catholique de Louvain.

Günther, S. and Sunkle, S. (2012). rbfeatures: Feature-oriented programming with ruby, *Sci. Comput. Program.* **77**(3): 152–173.

Hallsteinsen, S., Hinchey, M., Park, S. and Schmid, K. (2008). Dynamic software product lines, *Computer* **41**: 93–95.

Hanumansetty, R. G. (2004). *Model based approach for context aware and adaptive user interface generation*, Master's thesis, Virginia Polytechnic Institute and State University.

Hauptmann, B. (2010). *Supporting derivation and customization of user interfaces in software product lines using the example of web applications*, Master's thesis, University of Augsburg.

Henricksen, K. and Indulska, J. (2006). Developing context-aware pervasive computing applications: Models and approach, *Pervasive Mob. Comput.* **2**(1): 37–64.

Hinchey, M., Park, S. and Schmid, K. (2012). Building dynamic software product lines, *Computer* **45**(10): 22–26.

Hirschfeld, R., Costanza, P. and Nierstrasz, O. (2008). Context-oriented programming, *Journal of Object Technology, March-April 2008, ETH Zurich* **7**(3): 125–151.

Holzinger, A., Geier, M. and Germanakos, P. (2012). On the development of smart adaptive user interfaces for mobile e-business applications - towards enhancing user experience - some lessons learned, *DCNET/ICE-B/OPTICS*, pp. 205–214.

IBM (2003). An architectural blueprint for autonomic computing, *Technical report*, IBM.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E. and Huh, M. (1998). Form: A feature-oriented reuse method with domain-specific reference architectures, *Annals of Software Engineering* **5**(1): 143–168.

Kang, K., Cohen, S., Hess, J., Nowak, W. and Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study, *Technical Report CMU/SEI-90-TR-21* .

Kästner, C. and Apel, S. (2009). Virtual separation of concerns - a second chance for preprocessors, *Journal of Object Technology* **8**(6): 59–78.

Kästner, C., Apel, S. and Kuhlemann, M. (2009). A model of refactoring physically and virtually separated features, *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, ACM, New York, NY, USA, pp. 157–166.

Kastner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F. and Apel, S. (2009). Featureide: A tool framework for feature-oriented software development, *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, IEEE Computer Society, Washington, DC, USA, pp. 611–614.

Kaviani, N., Mohabbati, B., Gasevic, D. and Finke, M. (2008). Semantic annotations of feature models for dynamic product configuration in ubiquitous environments, *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, in collaboration with ISWC 2008*, Karlsruhe, Germany.

Kim, J. and Lutteroth, C. (2009). Multi-platform document-oriented guis, *Proceedings of the Tenth Australasian Conference on User Interfaces - Volume 93*, AUIC '09, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 27–34.

Kramer, D., Clark, T. and Oussena, S. (2011). Platform independent, higher-order, statically checked mobile applications, *International Journal of Design, Analysis and Tools for Circuits and Systems* **2**(1): 14–29.

Kramer, D., Kocurova, A., Oussena, S., Clark, T. and Komisarczuk, P. (2011). An extensible, self contained, layered approach to context acquisition, *Proceedings of the Third International Workshop on Middleware for Pervasive Mobile and Embedded Computing*, M-MPAC '11, ACM, New York, NY, USA, pp. 6:1–6:7.

Kramer, D., Oussena, S., Komisarczuk, P. and Clark, T. (2013). Graphical user interfaces in dynamic software product lines, *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*, pp. 25–28.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80, *J. Object Oriented Program.* **1**(3): 26–49.

Krueger, C. W. (2002). Easing the transition to software mass customization, *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, Springer-Verlag, London, UK, pp. 282–293.

Lavie, T. and Meyer, J. (2010). Benefits and costs of adaptive user interfaces, *Int. J. Hum.-Comput. Stud.* **68**(8): 508–524.

Lee, J. and Kang, K. C. (2006). A feature-oriented approach to developing dynamically reconfigurable products in product line engineering, *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, IEEE Computer Society, Washington, DC, USA, pp. 131–140.

Lee, J., Kotonya, G. and Robinson, D. (2012). Engineering service-based dynamic software product lines, *Computer* **45**(10): 49–55.

Liang, S. and Bracha, G. (1998). Dynamic class loading in the java virtual machine, *SIGPLAN Not.* **33**: 36–44.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L. and López-Jaquero, V. (2005). Usixml: A language supporting multi-path development of user interfaces, *Proceedings of the 2004 International Conference on Engineering Human Computer Interaction and Interactive Systems*, EHCI-DSVIS'04, Springer-Verlag, Berlin, Heidelberg, pp. 200–220.

Lincke, J., Appeltauer, M., Steinert, B. and Hirschfeld, R. (2011). An open implementation for context-oriented layer composition in contextjs, *Science of Computer Programming* **X**: 19.

Malek, S., Esfahani, N., Menasce, D. A., Sousa, J. P. and Gomaa, H. (2009). Self-architecting software systems (sassy) from qos-annotated activity models, *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, IEEE Computer Society, Washington, DC, USA, pp. 62–69.

Marcos, E. (2005). Software engineering research versus software development, *SIGSOFT Softw. Eng. Notes* **30**: 1–7.

Marinho, F., Lima, F., Ferreira Filho, J., Rocha, L., Maia, M., de Aguiar, S., Dantas, V., Viana, W., Andrade, R., Teixeira, E. and Werner, C. (2010). A software product line for the mobile and context-aware applications domain, *in* J. Bosch and J. Lee (eds), *Software Product Lines: Going Beyond*, Vol. 6287 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 346–360.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P. and Cheng, B. H. C. (2004). Composing adaptive software, *Computer* **37**(7): 56–64.

Moisan, S., Rigault, J.-P. and Acher, M. (2012). A feature-based approach to system deployment and adaptation, *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, pp. 84–90.

Myers, B. A. (1988). A taxonomy of window manager user interfaces, *IEEE Comput. Graph. Appl.* **8**: 65–84.

Nicols, J. (2006). *Automatically Generating High-Quality User Interfaces for Appliances*, PhD thesis, Camegie Mellon University.

Nielsen, J. (1990). Designing user interfaces for international use, Elsevier Science Publishers Ltd., Essex, UK, chapter Usability Testing of International Interfaces, pp. 39–44.

Parnas, D. L. (1976). On the design and development of program families, *IEEE Trans. Softw. Eng.* **2**: 1–9.

Parra, C. (2011). *Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations*, PhD thesis, INRIA Lille Nord Europe Laboratory.

Parra, C., Blanc, X. and Duchien, L. (2009). Context awareness for dynamic service-oriented product lines, *SPLC '09: Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, Pittsburgh, PA, USA, pp. 131–140.

Paskalev, P. (2009). Rule based gui modification and adaptation, *Proceedings of the International Conference on Computer Systems and Technologies and Workshop* *for PhD Students in Computing*, CompSysTech '09, ACM, New York, NY, USA, pp. 93:1–93:7.

Paskalev, P. and Nikolov, V. (2004). Multi-platform, script-based user interface, *Proceedings of the 5th international conference on Computer systems and technologies*, CompSysTech '04, ACM, New York, NY, USA, pp. 1–6.

Paymans, T. F., Lindenberg, J. and Neerincx, M. (2004). Usability trade-offs for adaptive user interfaces: ease of use and learnability, *Proceedings of the 9th international conference on Intelligent user interfaces*, IUI '04, ACM, New York, NY, USA, pp. 301–303.

Pleuss, A., Hauptmann, B., Dhungana, D. and Botterweck, G. (2012a). User interface engineering for software product lines: the dilemma between automation and usability, *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '12, ACM, New York, NY, USA, pp. 25–34.

Pleuss, A., Hauptmann, B., Keunecke, M. and Botterweck, G. (2012b). A case study on variability in user interfaces, *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, ACM, New York, NY, USA, pp. 6–10.

Pohl, K., Böckle, G. and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*, 1 edn, Springer.

Pohl, K., van der Linden, F. and Metzger, A. (2006). Software product line variability management, *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, IEEE Computer Society, Washington, DC, USA, pp. 219–.

Rabiser, R., Heider, W., Elsner, C., Lehofer, M., Grünbacher, P. and Schwanninger, C. (2010). A flexible approach for generating product-specific documents in product lines, *in* J. Bosch and J. Lee (eds), *Software Product Lines: Going Beyond*, Vol. 6287 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 47–61.

Rodríguez-Gracia, D., Criado, J., Iribarne, L., Padilla, N. and Vicente-Chicote, C. (2012). Runtime adaptation of architectural models: an approach for adapting user interfaces, *Proceedings of the 2nd international conference on Model and Data Engineering*, MEDI'12, Springer-Verlag, Berlin, Heidelberg, pp. 16–30.

Rosenmuller, M. (2011). *Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines*, PhD thesis, Otto-von-Guericke-University Magdeburg.

Rosenmüller, M., Siegmund, N., Apel, S. and Saake, G. (2011a). Flexible feature binding in software product lines, *Automated Software Engg.* **18**(2): 163–197.

Rosenmüller, M., Siegmund, N., Pukall, M. and Apel, S. (2011b). Tailoring dynamic software product lines, *SIGPLAN Not.* **47**(3): 3–12.

Rosenmüller, M., Siegmund, N., Saake, G. and Apel, S. (2008). Code generation to support static and dynamic composition of software product lines, *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, ACM, New York, NY, USA, pp. 3–12.

Russo, P. and Boor, S. (1993). How fluent is your interface?: Designing for international users, *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, ACM, New York, NY, USA, pp. 342–347.

Savidis, A. and Stephanidis, C. (2010). Software refactoring process for adaptive user-interface composition, *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '10, ACM, New York, NY, USA, pp. 19–28.

Schaefer, I., Bettini, L., Damiani, F. and Tanzarella, N. (2010). Delta-oriented programming of software product lines, *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, Springer-Verlag, Berlin, Heidelberg, pp. 77–91.

Schlee, M. (2002). *Generative programming of graphical user interfaces*, Master's thesis, University of Applied Sciences of Kaiserslautern.

Schlee, M. and Vanderdonckt, J. (2004). Generative programming of graphical user interfaces, *Proceedings of the working conference on Advanced visual interfaces*, AVI '04, ACM, New York, NY, USA, pp. 403–406.

Schuster, C., Appeltauer, M. and Hirschfeld, R. (2011). Context-oriented programming for mobile devices: Jcop on android, *Proceedings of the Workshop on Context-Oriented Programming (COP) 2011*, Lancaster, UK.

Sinnema, M., Deelstra, S., Nijhuis, J. and Bosch, J. (2004). Covamof: A framework for modeling variability in software product families, *Software Product Lines*, Vol. 3254 of *Lecture Notes in Computer Science*, Springer, pp. 197–213.

Sottet, J.-S., Calvary, G., Coutaz, J. and Favre, J.-M. (2008). A model-driven engineering approach for the usability of plastic user interfaces, *in* J. Gulliksen, M. B. Harning, P. Palanque, G. C. Veer and J. Wesson (eds), *Engineering Interactive Systems*, Vol. 4940 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Heidelberg, pp. 140–157.

Stuerzlinger, W., Chapuis, O., Phillips, D. and Roussel, N. (2006). User interface facades: towards fully adaptable user interfaces, *Proceedings of the 19th annual ACM symposium on User interface software and technology*, UIST '06, ACM, New York, NY, USA, pp. 309–318.

Sutton, S. M. and Rouvellou, I. (2004). *Concern Modeling for Aspect-Oriented Software Development*, Aspect-Oriented Software Development, Addison-Wesley Professional, chapter 21, pp. 479–505.

Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S. M. (1999). N degrees of separation: multi-dimensional separation of concerns, *Proceedings of the 21st international conference on Software engineering*, ICSE '99, ACM, New York, NY, USA, pp. 107–119.

Thüm, T. (2008). *Reasoning about feature model edits*, Master's thesis, Otto-von-Guericke-University Magdeburg.

Thum, T., Kastner, C., Erdweg, S. and Siegmund, N. (2011). Abstract features in feature modeling, *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, IEEE Computer Society, Washington, DC, USA, pp. 191–200.

Vanderdonckt, J., Calvary, G., Coutaz, J. and Stanciulescu, A. (2008). Multimodality for plastic user interfaces: Models, methods, and principles, *in* D. Tzovaras (ed.), *Multimodal User Interfaces*, Signals and Commmunication Technologies, Springer Berlin Heidelberg, pp. 61–84.

Voelter, M. and Groher, I. (2007). Product line implementation using aspect-oriented and model-driven software development, *Proceedings of the 11th International*

Software Product Line Conference, SPLC '07, IEEE Computer Society, Washington, DC, USA, pp. 233–242.

Weiser, M. (1991). The computer for the 21st century, *Scientific American Special Issue on Communications, Computers, and Networks*.

Weiss, D. M. and Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Zave, P. (2003). An experiment in feature engineering, *in* A. McIver and C. Morgan (eds), *Programming methodology*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 353–377.