Software Architecture Research and Development Project Synopsis

Building product lines from existing software

Department of Computer Science, University of Aarhus

Aabogade 34, 8200 Århus N, Denmark

Group: Bravo

20074842, Lars Kringelbach, lars@kringelbach.com

20064684, Marjus Nielsen, Marjus.nielsen@gmail.com

20074877, Morten Herman Langkjær, morten.herman@gmail.com

20054680, Peter Madsen, pm@chora.dk

<<Date: August 28, 2008>>

Abstract

This document contains the synopsis for an architecture research and development project, which is the final part of the Master in IT course "Software Architecture in Practice". The main focus of the project will be the task of converting existing software into a software product line.

Table of Contents

1 Motivation

In many software development organizations, the different products developed share common characteristics. For organizations specialized in a specific field of development the similarities seems to often outnumber the differences and this applies to several types of assets, such as requirements, design and implementation.

A software product line is defined as strategic reuse of all these assets in order to produce a family of products [Bass et al., 2003]. Developing products from a product line instead of making individual products can have several advantages such as a lower cost, shorter time-to-market, and higher productivity, which improve a company's ability to stay on the market with lower prices or have greater revenue and lessen the need for hiring more qualified people which are a limited resource. These advantages will however only apply when the product line is successfully implemented, which includes both a software architecture that will fit the company's needs but also an organization that is structured to handle the use of software product lines.

The project participants all agree that an existing project within their working environment could benefit from being converted to a product line, thus increasing development efficiency and company profits, but have lacked the experience, argumentation and support to pilot such a project within their working organizations. Thus it is the aim for the group to examine the subject of product line conversion in sufficient detail to be able to pilot similar conversions in their respective organizations. It is also the hope that this project can be used as inspiration for people aiming to convert existing software to a product line.

2 Hypothesis/Problem statement

The project will examine theory and practices supporting product lines in order to get hands on experience and to discuss conclusions in current theory.

A lot of software is written for dedicated purposes with no or little support for reuse in other projects, and will often result in source code being copied between projects instead of being reused in a strategic manner.

With focus on the software architecture of a software product line, the scope of this report is to discuss the principles behind, and to show samples of applying architectural patterns and other architectural relevant concepts supporting product lines.

These theories will be applied in practice in order to show how the architecture of an existing software product can be converted into an architecture that supports multiple similar products and hence strategic reuse of source code.

In practice the project will be founded in using an existing software product as a case for trying product line practices. As a case the open source project *DrinkMixer* [1] will be used. Some architectural reconstruction of *DrinkMixer* will be necessary, but the effort will be kept at a minimum and an in-depth discussion on this subject will not be part of the report.

Please note that the practical focus will be on the architectural aspects of software product lines. Other related assets, e.g. requirements, analysis and testing will not be taken into account. No practical work will be done regarding management related assets such as plans and budgets or in assessing an organizations readiness for implementing product lines. The theory section and to some extent the evaluation will cover the theoretical aspects of organizational readiness.

3 Method

In order to examine theory and practices, a presentation of the major theory[1] on product lines will be created. Special focus will be on presenting architectural patterns supporting product line development.

When techniques and patterns have been established, the current architecture of the case project will be identified. As stated above some architectural reconstruction will be needed. This process will focus on building some static views, mainly module- and Component & Connector views, in order to document the architecture 'as implemented'. *DrinkMixer* is limited in size though, so this process is expected to consume only a small part of the project effort.

[1] Currently [2], [3] and [4] are identified, but an analysis of available literature is expected to extend this list.

Once the current architecture is documented in sufficient detail, it will be redesigned to support product development with major reuse of source code in order to fit into a product line.

A software architecture used in a product line need often support behavioral differences between products opposed to only being configurable by parameterization. The analysis of *DrinkMixer* might show that it lacks sufficient business logic to be a well suited candidate for showing this kind of configuration. If this should be the case, some suitable functionality will be added (e.g. a pricing model, printing etc.).

This work will be based on using architectural prototypes, both presenting the theory behind these and to build one supporting a limited number of variability points.

The work needed to build new products using the product line will be assessed in order to comment on the overhead involved in using a product line strategy as opposed to creating individual products. The comparison will consider the percentage of files and lines that are different between products.

The product line architecture should at least support the same quality attributes as the original product it is being converted from, and have a higher degree of modifiability. Since the original quality attributes are unknown and rediscovering them from the source code is time consuming, we will restrict this portion of the project to a best effort guess.

4 Theory

4.1 Product lines

4.1.1 Introduction

What is a product line?

[Clements and Northrop, 3] defines a product line as:

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

According to this definition the products that a product line consists of are *“software-intensive”*. It must be argued that this definition must also be true for systems having nothing at all to do with software. In fact many historians trace the product line concept to the way that Eli Whitney manufactured rifles using interchangeable parts. This was in the early 1800s, long before the software was used.

The scope of this paper is software product lines and thus the definition used throughout the paper will be as [Clements and Northrop, 3] defines.

The most important fact in this definition is that it specifies that systems are developed from a common set of core assets in a prescribed way. The fact that the products need to have a set of core assets that are common to all of them means that a set of similar products developed separately are not a product line but only a set of similar products.

In addition to this the definition states that a product line must target a specific market segment or mission. This requirement is discussed later in the chapter on ‘Product Line scoping’.

What is not a product line

A good way to clarify what a product line is, is to describe what is not, so the following is a brief description of software development approaches, which are often mistaken for product lines:

Component-based or service-based development only

Even though many software product lines rely on a form of component-based (or service-based) development, this type of development doesn't make a product line. In order to become a product line these components must all be specified by the product line architecture. The components must also contain variation points making them adaptable to specific products.

Single-systems with reuse

This approach is about cloning an existing system (e.g. making a copy of the source code etc.) modify and add to it as needed to create the new product. Clearly a part of another system has been reused, but it is not a product line, but two different systems, not two systems built from the same core assets. It is the experience of the authors of this paper that this approach is widely used and might be viable in some scenarios. A discussion of pros and cons will be part of the discussions section at the end of this paper.

Releases and versions of single products

New versions and releases of a product are typically based on the same assets, architecture, components, tests etc. from earlier releases. This is in contrast to a software product line, which has multiple simultaneous products most likely having their own release and version plans.

Component level reuse

The opportunistic reuse of small pieces of code, algorithms, modules etc. stored in a common library shared by developers, as opposed to a product line where the reuse is planned, enabled, and enforced.

Development standards

Development standards might specify the choices that can be made. E.g. the forced use of a specific framework, database server, vendor etc. A product line might need to conform to such standards, but they are simply constraints not a product line.

Core assets

The core assets are reusable artifacts and resources that form the basis for the software product line. The following are typical examples of core assets:

- The architecture
- Performance measurements and real-time modeling
- Reusable Components
- Requirements
- Documentation and specifications
- Test plans and test cases

The list above shows the most prominent examples only and many other artifacts might be considered as core assets. Other assets might include performance models, COTS software etc. Management related items such as plans and budgets might also be candidates.

These other artifacts will not be considered as part of this paper.

The following is a short discussion of the different artifacts and how they relate to the software product line.

Architecture

The architecture represents a large investment from the architects, which are a limited resource in many organizations, so using the same architecture for multiple products will put less strain on a typically limited resource.

The required qualities like performance, reliability, modifiability, security etc. are a result of specific architectural decisions and tactics and as a result of this these qualities are anchored in the architecture. The specific architecture is used for all products and as a result of this they get the same qualities, thus minimizing risks.

Please note that quality attributes might be varied between products in a product line, e.g. the basis version might only support a single processing unit, while the enhanced version supports multiple processing units thus enhancing performance. The difference here is that the specific product line architecture supports variability on specific qualities, enabling different levels of quality in the individual products, but the quality of the product line architecture itself remains constant.

Performance measurements and real-time modeling

Existing measurements of performance associated analyses are candidates for product line core assets. When building new products from the product line, these measurements give high confidence that any timing-, concurrency-, network load- and deadlock issued have been eliminated.

Reusable Components

A high percentage (Up to 100%) of the components in the core asset base is used in the individual products. Components might require some alterations to fit the specific product, but the design, data structures and algorithms of the components in the core assets base is intact.

Product specific components needs development in order to provide functionality specific to the individual products, but these will be product specific assets, not core assets and will follow a lifecycle mainly independent of the core components. (The lifecycle of product lines will be discussed in detail later).

Requirements

A set of common requirements for the product line is a core asset as they are applicable for all products. In addition to these product specific requirements might exist, stating where the requirements for the specific product differ from the common requirements. Getting the major parts of the requirements from an already established common base means that feasibility studied etc. are automatically inherited, leaving only the analysis of the product specific requirements.

Documentation and specifications

As the full architecture and the common components are inherited in individual products, any documentation and specifications regarding these artifacts are inherited to, and thus should be part of the core assets. The benefit when building a specific product is that only product specific functionality and artifacts needs documentation work.

Test plans and test cases

During the development of the product line, testing artifacts like test plans, test cases, test data, test harnesses, integration- and unit tests are (or at least should have been) developed. Means to report and fix problems have already been used and proved. When building a new product the testing artifacts only need to be tailored to the product specific variations.

4.1.2 Cost and benefits of product lines

As seen in the sections above product lines introduces some overhead by preparing and maintaining core assets and creating an architecture tailored for variability, but it also introduces savings in time and effort when building additional products. Figure 4-1 from [McGregor, Northrop, Jarrad & Pohl, 4] presents an analysis showing that the total cost of building a product line as opposed to building individual products normally breaks even when building the third product, meaning that if only building two products, using an individual product strategy will most likely be the most cost effective and when building four or more the product line approach will most likely be the most cost effective. This number is of cause a typical number calculated as a mean from multiple projects, and special circumstances might set the break even higher or lower.

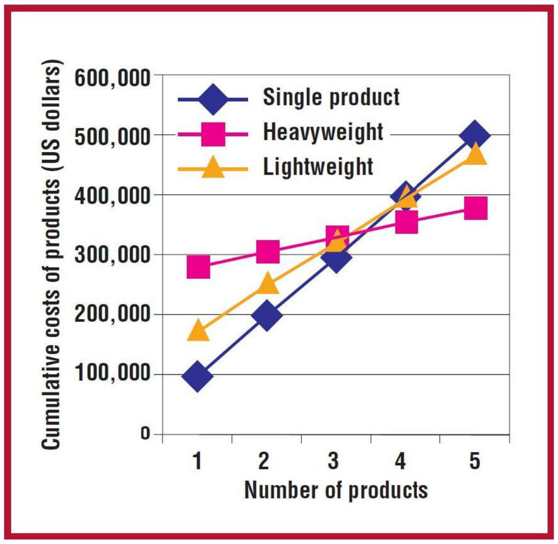


Figure 4-1 - Benefits of product lines

The figure shows the cumulative costs for individual product development (blue line) and two different product line development strategies (yellow- and pink lines). As the figure shows two different approaches to product lines exists as described below.

Heavyweight

A heavyweight product line is based on a proactive strategy where the reusable core assets are created before any products. The product line is designed and build from the ground up, with no constraints from existing products.

Lightweight

A lightweight product line is based on a reactive strategy, where core assets are extracted from one ore more existing product(s). As a result of this the product line will be constrained by the assets of the donor project(s), which might limit the quality of the resulting product line. The effort in building the product line will be less using this approach, but so is the potential reuse.

An interesting conclusion can be drawn from the figure. It appears that a lightweight approach will never be the optimum solution. When building less than three products, the figure shows that the single product approach is the most cost effective; above three the heavyweight product line approach is the most effective, meaning that the lightweight solution is never the optimal solution.

The following is a short presentation of the costs and benefits normally associated with product lines.

Costs

The costs in building a product line comes down to building the core assets. These assets are also needed for individual products (e.g. an individual product also need an architecture, requirements etc.), but extra work will need to into building these, in order to make them sufficient for supporting a product line. In addition to this the work of tailoring the individual will be added to the total cost.

Benefits

Main parts of the assets needed to build a specific product come from the product line core assets, reducing the cost of producing a product when a product line is in place. As seen in the figure above the total savings increase with the number of products build. In addition to benefits on the project level, the case studies in [Clements and Northrop, 3] shows that organizational level benefits can also be achieved including:

- Ability to sustain unprecedented growth
- Decreased time to market
- Decreased product risk
- Increased market agility
- More efficient use of human resources

- Increased product quality

Cost and benefits of specific core assets

The following table lists core assets and their benefits for a product line and the added costs they will put on the product line.

Asset	Cost	Benefit
Architecture: The architecture for the product line.	The architecture must support the variation required by all products in the product line. This adds additional requirements to the architecture and adds to the efforts and skills required by the architect.	The architecture represents a large investment from the architects, so using a product line architecture for multiple products will put less strain on a typically limited resource, as the architecture is largely completed for subsequent products.
Performance measurements and real-time modeling: Products with real-time or near real-time performance requirements, performance measurements and real-time modeling are needed to ensure that requirements are met.	If these artifacts are to be reused, it puts the constraint on the product line that variability cannot exist where performance and real-time behavior is affected. If in doubt a new analysis of (parts of) the product line needs to be performed.	When building new products from the product line, these measurements give high confidence that any timing-, concurrency-, network load- and deadlock issued have been eliminated.
Components: The software components are the building blocks for the products in the product line.	The components must be designed for reuse. They must be robust and should support variability points and be easily extensible. In order to make components fit multiple products they also need to be general in nature. All in all these additional requirements adds to the development time of the core components.	Typically the interfaces are reused directly, as are most of the components.
Requirements: Common requirements for the product line. In addition to these product specific requirements might exist	Specifying the requirements for a product line will require an extra analysis effort, and it might be difficult to conclude on both common requirements and variation points that are applicable to all (planned) products.	Common requirements and variation are documented explicitly, which aids in developing the product line architecture. When building new products, they will be much simpler to specify, due to reuse of large parts of the requirements.

Documentation and specifications: Documentation for the product line architecture and components.	The product line must be well documented, including additional subjects like scoping, use of variability points etc. adding to the effort needed.	The benefit when building a specific product is that only product specific functionality and artifacts needs documentation work.
Test plans and test cases: Testing artifacts for the entire product line.	All artifacts must support more than one product and support variation among the products.	Testing artifacts like test plans, test cases, test data, test harnesses, integration- and unit tests have already been developed. When building a new product the testing artifacts only need to be tailored to the product specific variations.
People and skills: All staff working on the individual products is also working on the product line.	All staff must understand software product line practices and how to use the core assets. This might require special training and extra documentation and specifications.	The individual products have so much in common that staff working on one product can easily work on another. Even though a product line approach might require extra training, the knowledge gained can be used when developing subsequent products.

Table 4-1 Product line core assets

4.1.3 Building product lines

Three main activities take place when building a software product line:

- Core assets development
- Developing the products using the core assets
- Management

The order in which the activities are performed depends on whether a lightweight or heavyweight approach is used.

Even when using a heavyweight approach, development of specific products might result in adjustments to the product line architecture.

Core assets development

The core assets are build to facilitate developing products. As described in [Clements et al. 2002] this is an iterative process, where the core asset might influence the context it was build from, e.g. changing the product line scope (se the section below for details) will change the context of which classes of systems to analyze.

The main contexts constraining core asset development are:

Product constraints

Common features and variations in the products, future feature requirements, standards to follow, interfaces to existing systems, quality requirements etc.

Production constraints

Time to market, production capability, standards to follow etc.

Production strategy

Heavyweight or lightweight, in-house development or use of COTS components, automatic generation of products or manual assembly etc.

Preexisting assets

Legacy systems, existing products, libraries, frameworks, algorithms, tools, components etc.

Scoping

The scope of a product line defines which products are to be build as part of the product line. What commonalities do the products have and what are the variations. Normally a feature is clearly part of the product line and other functionalities are clearly not. In between these boundaries a "grey area" exist, where the decision whether to include the feature or not is decided on a case by case basis. The decision to include "gray-area" functionality will typically be motivated by the wish to enter new markets, measure up to the competition etc.

When discussing product line scope, the literature distinguishes between narrow- and broad scope product lines as follows:

Narrow scope

Narrow scope product lines vary only in a relatively small number of features, and are good candidates for specification of new products by means of specialized tools or domain specific languages. In the extreme case individual products can be build by configuration only. Specifying too narrow a scope might be a problem as too few products can be build from the product line to justify the effort.

Broad scope

Broad scope product lines vary both in kind and in features. This gives the possibility of building many different products from the same product line. Broad scoped product lines are typically build as frameworks or a set of services. Specifying too broad a scope can also be problem as the effort needed to build the product line increases when the scope broadens, e.g. building an architecture that supports multiple products with a large number of features, will most likely require a large effort and result in a complex architecture. The task of building the product line might end up being so big that nothing is gained.

Selecting a scope

As seen above getting the scoping for a product line correct is very important for achieving success. As stated in [Bass et. al 2003] it is easy for an architect to find commonalities between systems, but the real problem when scoping is to find the commonality that can be used to reduce costs in a substantial way.

Other items than the products themselves should be considered when scoping. Thinks like market segmentation and target users might be used to determine product lines. E.g. Sony Ericsson has a Walkman phone product line targeted towards young people and a Personal Information Manager phone product line targeted towards business use.

The scope of the product line is not static but will typically evolve as market conditions change, as new opportunities arise, or as the capabilities of architects and tools evolve.

Building the core asset base

The core asset base is the basis for the production of products in the product line. It might not be that all core assets are used in all products, put to qualify as a core asset the artifact must be used in a sufficient subset of the products to justify the effort gone into making it. [Clements et al. 2002] advises that each core asset should have a process description on how they are to be used when building products.

As stated earlier several artifacts can go into the core asset base, and the scope has just been identified as important. In addition to the scope, the architecture is central to the product line.

Product line architecture

The product line architecture plays a key role in the product line, as software architecture supports concepts of having constant as well as varying functionality, which is the essence of a software product line. It is the responsibility of the product line architecture to identify the allowed variations as well as providing mechanisms for achieving them. The variations might be extensive as the products might vary in behavior, quality attributes, platform etc. As stated by [Bass et. al 2003] the tasks of the architect when designing an architecture for a product line are:

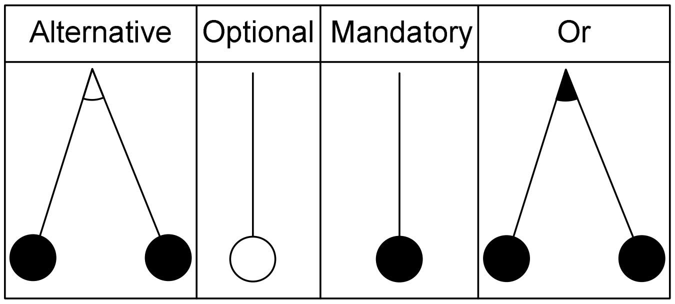
- Identifying variation points
- Supporting variation points
- Evaluating the architecture

Identifying variation points

Variation points might be identified at various stages of the product line development, typically during requirement specification and architecture developments, but some might also be identified during implementation, even when building subsequent products.

Some variation points might be interdependent, e.g. the user interface might be tied to the platform, using Windows Mobile as the platform will require one user interface and Windows Vista another.

[Kang et al, 1990] describes a method for feature analysis, and feature modeling where the variations of features in a product line are given by the variation of features across specific products in the product line. The way the variation is documented is by the notation shown in 4-2.



4-2 Feature modelling notation

The notation include the values and descriptions provided by Table 4-2:

Notation	Description
Alternative	Alternative represents mutually exclusive features where only one of more features can exist in a product at a time.
Optional	Optional features represent features that for different products can be either included or excluded.
Mandatory	Mandatory features represents features that will

	exist in all products of a product line
Or	"Or" features represents cases where for feature A & B, the product can exist with either feature A, or feature B or both.

Table 4-2 Explanation of Kang et al feature modelling notation

An example of how to use this has been taken from [Kang et al, 1990], and modified to show the Or notation. This can be seen in Figure 4-3

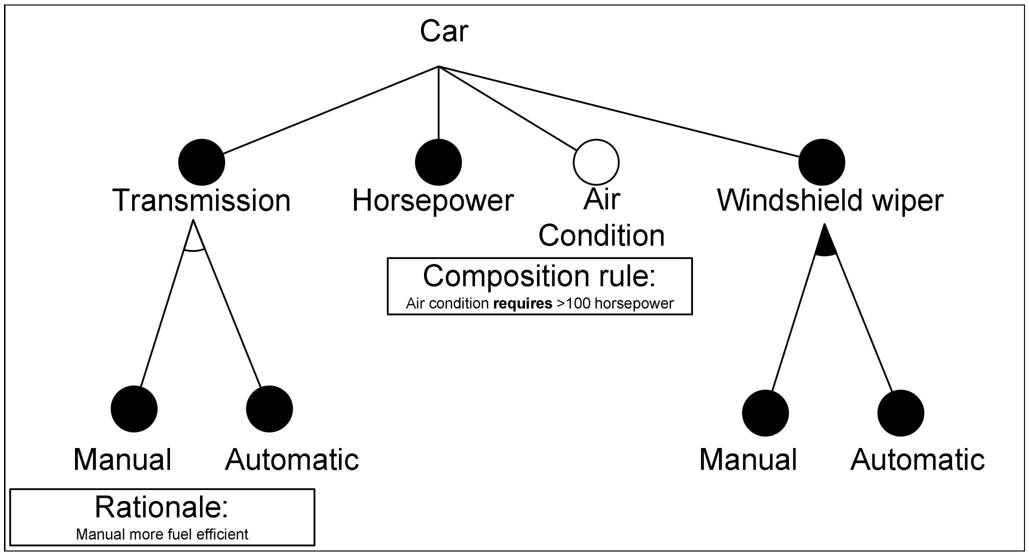
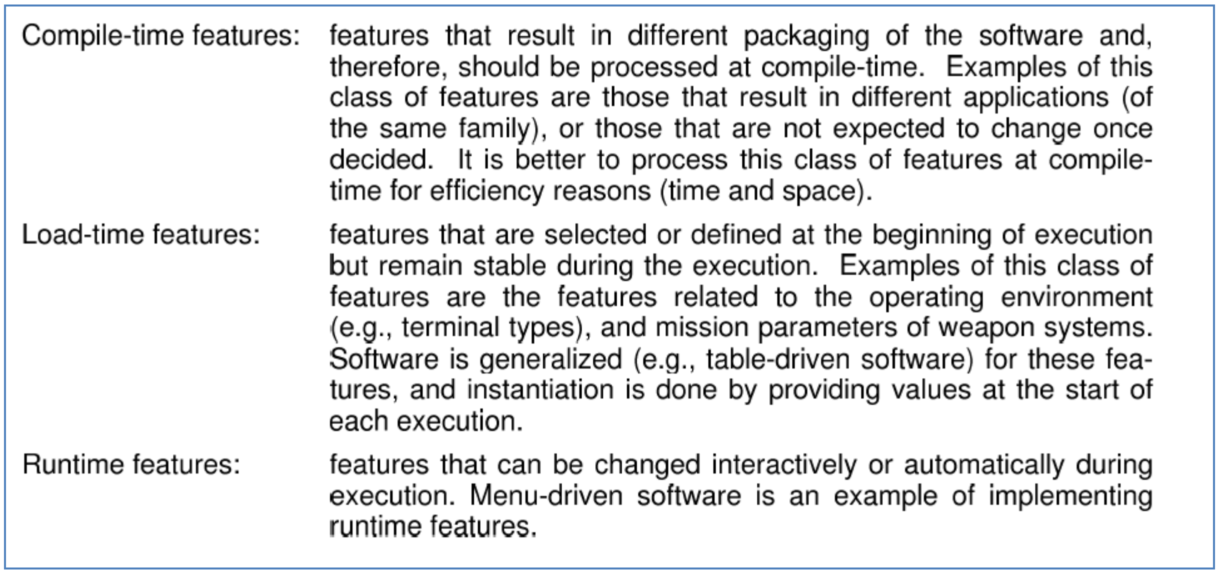


Figure 4-3 Feature modelling sample

In
Figure 4-3 the sample shows a car, that must have a transmission, and the transmission can either be manual or automatic. The sample also shows that a car always has horse powers, and that a car can optionally have Air condition. The Air condition feature is further constrained by the amount of horsepower included in the car. The car also has windshield wipers, and these can be automatic with rain sensor, manual or both, where it is possible for the driver to override the automatic sensor by using the windshield wiper manually.

Furthermore [Kang et al, 1990] categorizes the feature variations into Compile-time features, Load-time features and Runtime features. The explanation of the categories can be seen in Figure 4-4, which is captured from [Kang et al, 1990] page 43.



Compile-time features:	features that result in different packaging of the software and, therefore, should be processed at compile-time. Examples of this class of features are those that result in different applications (of the same family), or those that are not expected to change once decided. It is better to process this class of features at compile-time for efficiency reasons (time and space).
Load-time features:	features that are selected or defined at the beginning of execution but remain stable during the execution. Examples of this class of features are the features related to the operating environment (e.g., terminal types), and mission parameters of weapon systems. Software is generalized (e.g., table-driven software) for these features, and instantiation is done by providing values at the start of each execution.
Runtime features:	features that can be changed interactively or automatically during execution. Menu-driven software is an example of implementing runtime features.

Figure 4-4 Kang et al catogorization of features.

Supporting variation points

Multiple techniques for supporting variability exits as discussed in [Bass et al. 2003]:

- Inclusion or omission of elements by parameterization of build procedures.
- Using a different number of repeatable elements, e.g. increasing performance by adding mere servers.
- Programming to an interface and then changing the implementation to achieve the desired availability. Selection of the specific implementation can either be done at compile time or at run time by means of some configuration. This can be done by means of static- or dynamic link libraries.

The techniques above are changes on the architectural level, but variability can also be build into individual elements:

- Specialization and generalization of classes
- Overloading methods
- Building extension points into components.
- Using build time parameters (e.g. compiler directives)
- Using reflection (requires programming language support)

Specific architectural patterns or styles support product line development (as do some design patterns). These are described later in this document.

Evaluating the architecture

As a number of systems rely on the product line architecture, the evaluation becomes very important. Evaluation can be done using normal evaluation techniques like ATAM, CBAM and aSQA.

When evaluating focus should be on the variability points, ensuring they are appropriate and supports the planned products. The different quality requirements must also be evaluated.

The product line architecture should of cause be evaluated prior to starting building products, if quality attributes for product varies substantially, specific evaluations might be needed.

When assessing whether to build a product that is in the "gray area" of scoping, the architecture could be evaluated in respect to its suitability to support this new product.

Management

Management is important when developing a product line. The required resources must be assigned, and the product line activities must be coordinated and monitored. It is essential that management at all levels is committed to building the product line.

Management at the organizational level must ensure that an organization capable of building a product line is in place, both in numbers and in skills. They also typically drive the scoping and release plans.

Technical management is responsible for the development of the core assets and products and that processes are followed.

Relationship with the Architecture Business Cycle

One of the main focuses of [Bass et al. 2003] is the Architecture Business Cycle or ABC, which talks about the software architecture being a result of technical, business and social influences, where these influences influence the architect and thus the architects work. The knowledge gained by the architect and the work produced will again influence the organization and so on. This cycle will keep running over time.

This also applies to product lines in particular, where the ongoing development of a product line with producing new products, adjusting the core assets due to the knowledge gained, which leads to new product etc., is an example of the architecture business cycle, where more knowledge on building product lines is gained by the organization over time. Educating staff in product line practices will also result in gained product line knowledge and acceptance within the organization, resulting in better education. Many of these cycle effects might exist when building product lines.

Influence might not always be positive. Problematic experiences with product lines might lead to the organization not supporting the effort wholeheartedly, resulting in more problems etc., so ABC effects are quite prominent when discussing product lines.

4.1.4 Organizational requirements

As stated in the Motivation section, organizational support is very important in order to build a successful product line. Management need to accept that building the product line adds a development overhead, resulting in not being able to deliver the first product as soon as normally expected when building the products individually.

As all initiatives that involve large parts of an organization, it is of greatest importance that the full organization understands and supports the effort, and that the organization does not give up at the first sign of problems. [Clements et al. 2002] recommends that an individual or group should be assigned as product line with the responsibility of keeping the organization pointed toward the product line goals, in order to keep the organization on track even if problems are encountered.

Collecting, maintaining and maturing the core assets require discipline and maturity from the organization. Assets needs to be managed, documented and process descriptions on their use need to be developed and documented. Not all organizations are capable of marinating a continuous high standard in these non-coding related fields.

A product line development needs a well trained staff, the skills of the architect are crucial here, as the architecture of a product line is different to single product architectures, and might potentially be large and complex if a broad scope product line is developed.

4.2 Architectural Prototypes

Architectural prototypes can be used to aid in the design and development of a new architecture by investigating architectural concerns from the stakeholders [Bardram et. al, 2004]. They are very cost-effective in exploring and investigating different architectural solutions, analyzing requirements, measure quality attributes etc. And they provide a possibility to discuss important aspects of the design in a language that can be understood by many stakeholders.

In general, prototypes are used to test and evaluate a design by trying it in practice instead of reviewing descriptions. They are defined in [Floyd, 1984] as executable systems that "involve an early practical demonstration of relevant parts of the desired software". I.e. they can be used to describe the effect of specific requirements to end users or other stakeholders by showing them how it will work out in practice.

4.2.1 Properties of Architectural Prototypes

An architectural prototype has certain properties that separate them from traditional prototypes [Bardram et. al, 2004]:

Architectural prototypes are constructed for exploration and learning of the architectural design space.

As one of the following properties state, the architectural prototypes do not necessarily provide any functionality. They are created to investigate the architecture and as stated in [Bass et al., 2003], architecture and functionality are orthogonal so they are used in order for architects to extend his knowledge so he can choose the best architecture for a system.

In this project, this is exactly what shall be achieved with the prototype. It shall be used to investigate the possibilities for making a system modifiable by adding variation points in order for it to be generic enough to be part of a software product line.

Architectural prototyping addresses issues regarding architectural quality attributes in the target system.

As described in [Bass et al., 2003], quality attributes can never be achieved in isolation in complex systems. Often the achievement of one quality will affect the achievement of other. Modifiability and performance are often contradictory requirements. Modifiability concerns decreasing the development and maintenance cost, which can be achieved with decoupled code with limited communication paths, information hiding, indirection, runtime configuration etc. which often degrades performance. Performance on other hand concerns handling events within the defined timing constraints. This is not always possible when modifiability tactics have been applied. With hard timing constraints it may be necessary to have duplicated information, direct object access and static configuration optimized for performance.

When designing a product line, it is important that we can achieve the level of modifiability that is needed to support a product line. The architecture shall support changing certain parts of the functionality without changing the architecture itself. The buildability of the system with regard to actually building a product from the software platform is also of great importance.

Architectural prototypes do not provide functionality per se.
As described above, architecture does not deal with functionality and therefore an architecture prototype can often be independent on the actual functionality of the system. This is one of the great benefits because it enables creation of architecture prototypes at a very early stage in the development process and keeps the cost low. It may however be necessary to implement some functionality in order to have a real life simulation, e.g. when measuring performance.

The architecture prototype in this project is developed to investigate modifiability and buildability which does not involve actual functionality of the system. It may however be necessary to implement some functionality to verify that it is possible to achieve the needed level of variation in the products.

Architectural prototypes typically address architectural risks.
As described above, quality attributes often gives contradictory requirements to the architecture and therefore pose an architectural risk. Architectural prototypes are useful for evaluating qualities and thereby address these architectural risks. The technical environment influences the choices of the architect. An architectural prototype can be used to evaluate new technical solutions which often pose architectural risks, e.g. in order to evaluate the performance of a new message system.

The architectural risks in this project consist in whether a certain level of modifiability can be achieved so that is will support development of several similar products as described above.

Architectural prototypes address the problem of knowledge transfer and architectural conformance.
In order to ensure that the system is built according to the designed architecture, an architectural prototype can be used as a skeleton

system where the functionality can be built upon. As an example, the Rational Unified Processes uses evolutionary architectural prototypes (described below) which further development are based on as a key artefact [Rational, 1998].

The prototype described in this report is used for architecture evaluation and not for knowledge transfer or architectural conformance. If the *DrinkMixer* product line was to be implemented, it would however be useful for demonstrating how the architecture can be implemented in order to achieve the required qualities.

4.2.2 Classification of Architectural Prototypes

As described above, RUP uses ***Evolutionary*** architectural prototypes. An evolutionary prototype is developed with the target system in mind and preferably functioning in the correct environment. They can be used in the implementation phase by evolving functionality on the prototype, thereby ensuring architectural conformance.

Exploratory architectural prototypes [Bardram et. al, 2004] on the other hand are used for exploration of the architectural design space and can often be thrown away when the results have been evaluated.

For evaluation of a designed architecture, ***Experimental*** architectural prototypes can be used to investigate certain design solutions with regard to the qualities that the system must fulfil.

In the Rational Unified Process, these different types of architectural prototypes are generally used in different stages of the development process. Exploratory prototypes are used to find architectural proposals in the early phases of the project (Inception/Elaboration). In the elaboration phase, experimental prototypes are used to measure whether the designed architecture is adequate with regard to the required qualities. The goal of the elaboration phase is to have an evolutionary prototype for a production-quality component that further development can be based upon.

Based on the architecture reconstruction of *DrinkMixer* and the defined quality attribute requirements, the architecture design of `DrinkMixer` will be modified to support needed variation points. An experimental architectural prototype will be used to evaluate if the architecture can fulfil the quality requirements.

4.3 Architectural patterns

Throughout the project various software disciplines that support product lines will be used. We expect to explore some patterns and architectural styles, which can support the transformation to a product line. It is expected that some of the styles and patterns will be tested for suitability by implementing an architectural prototype on the style or pattern.

Patterns of any kind are generic and well proven solutions to recurring design problems. Patterns can generically be described as a three-part schema that consists of a context, a problem and a solution [Buschmann, 1996].

Context

The context describes the design situations where the problems exist.

Problem

A description of a problem in a given context. The very essence of the problem should be described, as well as more specific issues or forces e.g. requirements and quality attributes for the system. As with quality attributes in general, the forces can contradict each other.

Solution

The solution should solve the problem by balancing the forces in the best way possible under the given context. The solution includes two aspects: The static aspects, that is the structure of the system, and the dynamic aspects, that is the runtime behavior. In other words these aspects can be documented with a module viewpoint and a component & connector viewpoint, respectively.

The solution does not necessarily solve all problems. The solution may focus on a part of the problems and leave others untouched. For instance the Model-View-Controller focuses on decoupling the user interface from the rest of the system, but does not address other qualities such as security or performance.

Summed up [Buschmann, 1996] define software architecture patterns as:

> *An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities,*

and includes rules and guidelines for organizing the relationship between them.

Software architecture patterns are high-level abstractions of the system that extract the essence of the solution. Patterns must thus be tailored for every specific use. [Buschmann 1996] describe a medium level pattern called design patterns, that specify the architecture for smaller units of the system. Examples of design patterns are the Publisher-Subscriber pattern and the Proxy pattern.

All patterns are used to achieve certain qualities. Since the design patterns are lower level than architectural patterns they more specifically address qualities such as modifiability, that is, the design patterns focus on smaller parts of the whole architecture. Thus, the design patterns can be used to address quality attribute scenarios by having a standardized way of incorporating a selection of tactics that give the system the wanted qualities.

Since patterns are well proven solutions to common problems and they are given names that are familiar to architects and developers, they are also a way of documenting the solution and help communicating the architecture to the stakeholders.

Possible patterns

We will concentrate on styles and patterns that may lower coupling in the system, and support product line development. Examples are repository style for decoupling persistence, and model-view-controller for low coupling of the user interface etc.

Model-view-controller

- Context

  Applications that have a dynamic user interface.

- Problem

  Systems often behave differently based on which role the users belong to. Super users have other rights than standard users have. Some standard users might be interested in different parts of the system than other users.

  Many of the changes to the core of a system are reflected in the user interface. New functionality requires new forms or menu items to be reached.

Often it can be necessary to use a system in different environments e.g. in a desktop version, and a web version.

And sometimes it’s just necessary to view information from different angles. All the different existing chart types in a standard spreadsheet are a good example of this, where one set of data can be viewed in a myriad of different ways.

- Solution

  The model-view-controller is used to lower the coupling between a model of a domain and the presentation. Or in other words decoupling the user interface and the rest of the system, by dividing the system into three components

  - The model that contains all data and business logic.
  - The view that displays the model to the user.
  - The controller that receives user input and manipulates the model accordingly.

  The user interface is made of the view and the controller.

Other patterns

In order to support several types of products, we may benefit from having a Meta model for the persisted data, in order to interpret the data in different ways, in different user domains.

4.4 Architectural reconstruction

4.4.1 The process

The process of architectural reconstruction is obtaining the documentation of the architecture for an existing system.

The basic process is to Extract information from the system, abstract the data, and finally present the results. An example of a formal process for this is the Symphony process created by Nokia [Deursen et al., 2004] .

The Symphony process identifies three different artifacts, that is used to determine the architecture. These are:

- source views
- target views

- hypothetical views.

The source views are information that can be gathered from the existing system, such as abstract syntax trees, or other source code information.

The target views, describes the system as implemented, by abstracting the information from the source views, and identifying architectural significant information.

Finally, the hypothetical views exist, where other types of information is gathered about the system. This could be existing architecture descriptions, which may, or may not be in synchronization with the systems current implementation. It could also be presentations of the system, or documentation that could provide information on how and why particular architectural decisions have been made.

4.4.2 Performing Architectural reconstruction

The typical architectural reconstruction has a driver, which is founded in some sort of problem that needs to be discovered and fixed. In this case the problem is creating a business case of founding a software product line based on the existing implementation of DrinkMixers

The process of architectural reconstruction is typically a manual, or in best cases a semi automatic process.

When using tool support for gathering information from a system, the typical extracted information is static structure, if the source code is available. Other types of tool support, Island grammar [Koschke, 2005] [Deursen et al., 2004], which could help identifying dependencies, and groupings of data structures and thus extract data from the system.

Discotect [Schmerl et al., 2006] is another tool, which support probing a system for architectural significant events, such as identifying call stacks of when e.g. sockets are created. This can give some information on e.g. sequences in the system, and help reconstruct C&C views.

4.4.3 Drawbacks

The main problem with automated tools for extracting information for architectural reconstruction is that the information is representing the system in all details, where software architecture is an abstraction and an overview of the system. Architecture is created to omit insignificant details, thus creating a gap between the expected result of automated reconstruction, and the actual needs. An example of this can be seen, in the [Appendix INDSÆT REFERENCE, og indlæg autogenrerede diagrammer], where Enterprise Architect from Sparx Systems has been used to auto generate diagrams from the DrinkMixer. The auto generated diagrams can be compared to the manually generated diagram that omits insignificant details. When omitting such details, an additional amount of textural information is required, to communicate the actual purpose of the diagrams. The information, which is left out here, would normally be documented in a system design, or an architecture document. The manually created static views, has been created by manually analyzing the source code, and abstracting the details.

4.4.4 Articles on automatic reconstruction

When looking into the articles written on the topic of automated reconstruction [Koschke, 2005], it can be seen that approximately 4/5 of all papers on automatic reconstruction of views and viewpoints are articles concerning module views. 1/5 of all articles are on C&C views, and only a single article is written on how to automatically generate allocation views. This indicates that the tool support for automatic generation of architectural views, that does not directly relate to the static representation of the source code are somewhat immature, and/or that the topic is of so great complexity, that it has simply proven extremely difficult to auto generate these results. The reason for this could possibly be that the meaning of architecture is to omit details, whereas the system represented as source code or binary files contains all details.

5 Case: DrinkMixer

5.1 Architectural Reconstruction of DrinkMixer

The main focus of the report is not to perform a full architectural reconstruction. The following diagrams show a module view, a C&C view, and an allocation View. The views have been manually reconstructed by inspecting the code. For comparison reasons, module views generated by Enterprise Architect have been added to the appendix, to show the difference in abstraction.

5.1.1 Module View

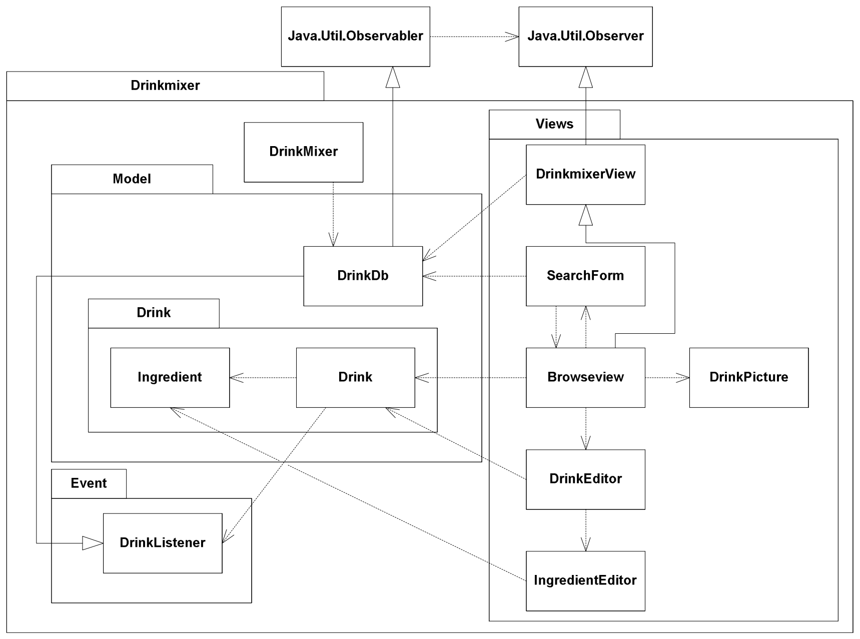


Figure 5-1 Module view of drink mixer

The module view shows the system module decomposition, and documentation of the system as implemented. However some details has been omitted. When analyzing the module view, it is clear that the decomposition of the implemented system lacks interfaces, and clear responsibilities, and abstraction at a required level to be highly modifiable. Modifiability is one of the key qualities in when creating a product line, so loose coupling is a premise for effectively testing the system in isolation, and enabling the process of creating new product with new functionality based on the same foundation of software assets.

5.1.2 C&C View

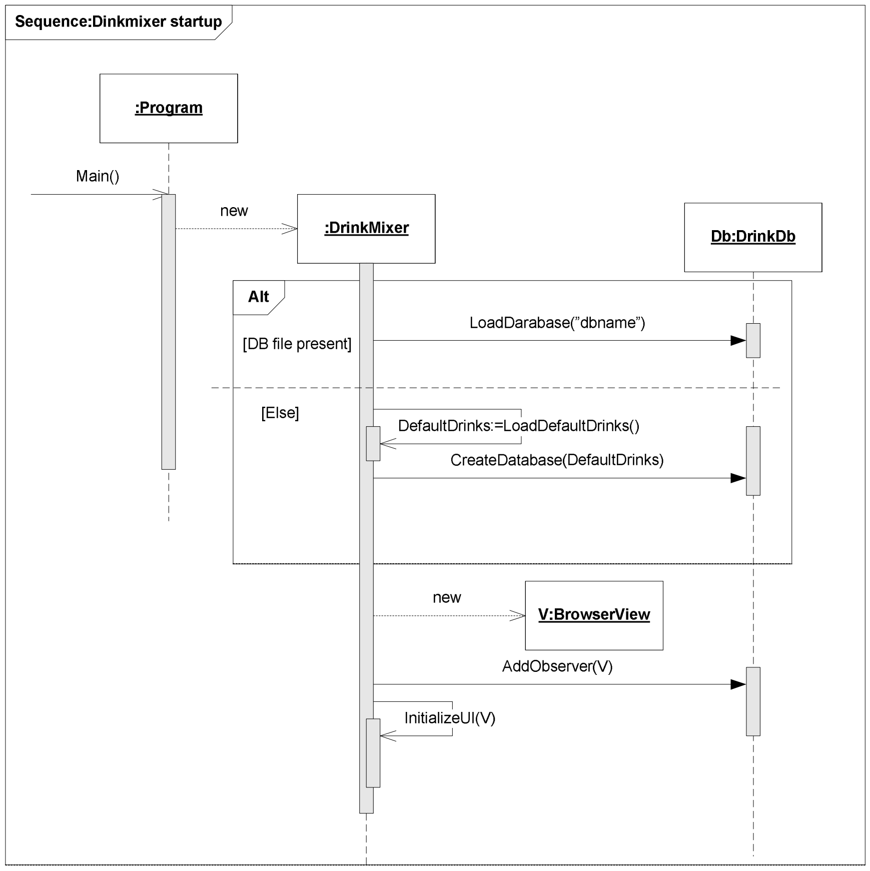


Figure 5-2 C&C sequence Drinkmixer startup

The sequence of startup shows how the view and database is coupled when starting the application. There are no interfaces, so it would be impossible without refactoring the system to change the implementation of the database, from a local flat file, to e.g. a remote Oracle 10G database.

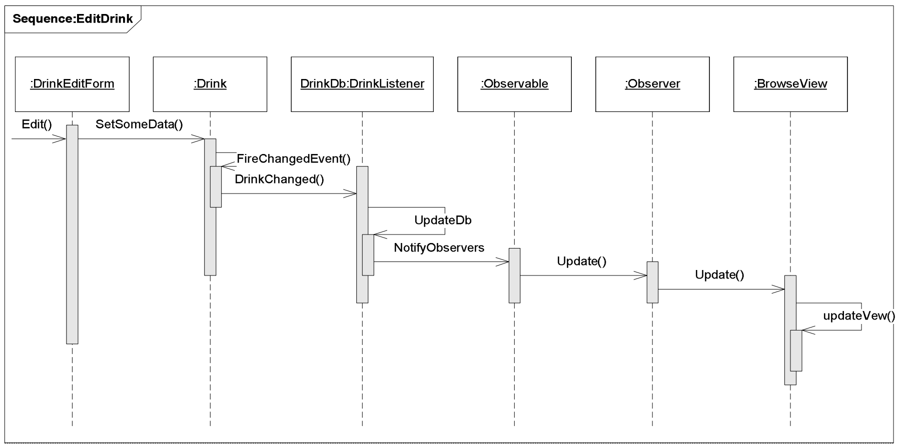


Figure 5-3 C&C Sequence edit drink

The sequence Edit drink, shows how the edit form, and drink communicate and persists updates to the database and furthermore updates the user interface through interface indirection, and through the observer pattern.

5.1.3 Allocation View

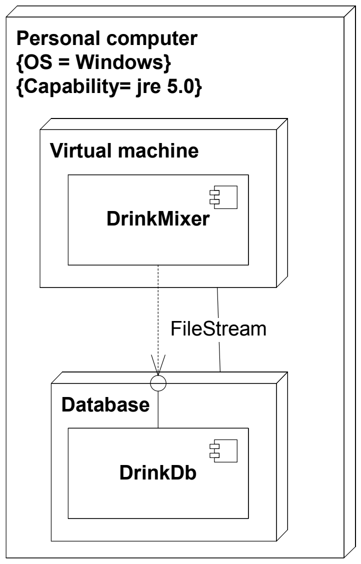


Figure 5-4 Allocation view of drink mixer

The allocation view shows the simple allocation of a client application, and a database, where the database is a flat file containing Java objects persisted through the standard java serialization API.

5.2 Product Line Definition

The product line developed as part of this project is somewhat special as it is a mix of a lightweight and a heavyweight approach. The product line starts out as lightweight due to the fact that assets are extracted from the existing product, *DrinkMixer*. As a result of this the product line will be constrained by the assets of this donor project. The functionality of *DrinkMixer* is limited and it is assessed that the product line that will come out of only using the lightweight approach will be to narrow and thus will result in to few products in the product line. In order to prevent this a heavyweight approach, stating functionality not part of the possible variations of *DrinkMixer* is performed (please note that *DrinkMixer* is only used as a case for product line development, so no reasoning for picking specific features will be presented, other than the fact that they make good candidates for variability).

The features were defined in a workshop where common features were defined and a feature model diagram was created as described below.

5.2.1 Common features

When building a product line, it is important to define both what is common and what varies.

During the workshop it was agreed that all products share the following aspects:

- They contain items, units and guides
- The guide instructs a user on how to assemble the unit using a set of items.
- All products have a Graphical User Interface (GUI)
- Items, units and guides are stored in some kind of storage

5.2.2 Building the feature model

One technique for specifying product line features is to build a feature model, describing mandatory and optional features, and thus specifying the variability possible for the product line. The specification is done using feature model diagramming as described in [Kang et al., 1990].

Due to the size, the model is divided into multiple sub diagrams, but is to be read as a whole (the full diagram can be seen in the Appendix section).

The following sections show the feature model diagram and outline the features. Please note that the features will not be described in the detail most likely needed for real product line use.

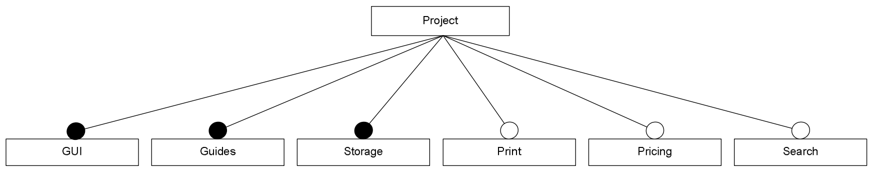


Figure 5 - Top level features

The figure above shows the top level features in the product line as explained below.

GUI

The GUI feature concerns user interface aspects. A GUI is mandatory for all Products.

Guides

The Guides are the instructions on how to perform the specific task (e.g. mixing a specific drink, performing a specific repair etc.). Guides are mandatory for all Products.

Storage

The Storage is the feature on which data are stored. A Storage system is mandatory for all Products.

Print

The Print feature concerns printing from the product. Printing is an optional feature, which might be omitted from some products.

Pricing

The pricing feature concerns price calculations on materials, labor etc. Pricing is optional.

Search

The search feature concerns search facilities on different types of data. Search is optional.

GUI Features

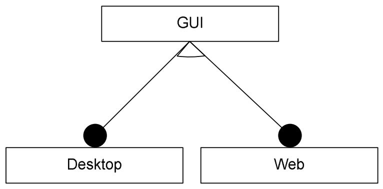


Figure 6 - GUI features

Two different GUI options will exist for the product line, either a desktop GUI or a WEB based thin client. Exactly one GUI type will exist in a specific product.

Guide Features

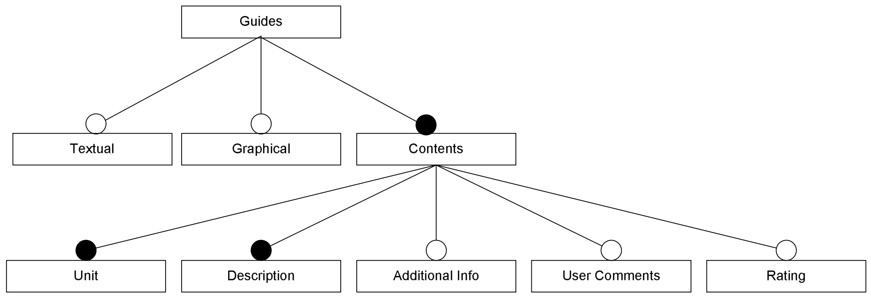


Figure 7 - Guides Features

Guides can optionally contain textual and/or graphical elements, but must contain some contents as described below.

Textual

Guide information written in clear text. The guide might not have any clear text (e.g. picture only guide).

Graphical

Pictures or illustrations guiding the user. The guide might not have any graphical elements.

Contents

The different types of contents that might be part of a guide.

Unit:

Which task / item does this guide concern. Mandatory.

Description:

The guiding itself. Mandatory.

Additional info:

Extra information applicable to the guide, e.g. legal info etc. Optional.

User comments:

The ability for users to enter their own comments and include it with the Guide. Optional.

Rating:

The ability for users to rate the item that the guide concerns, like rating a recipe form one to five chef's hats. Optional.

Storage Features

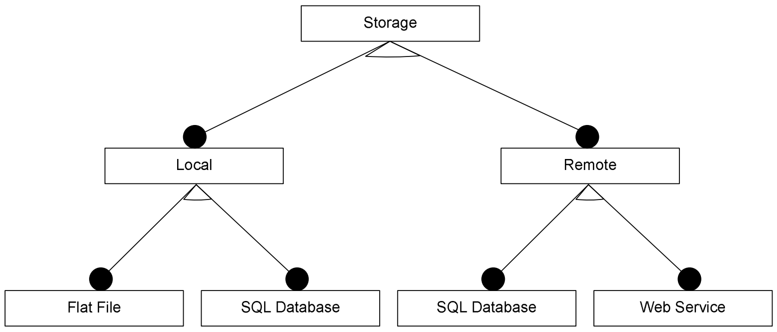


Figure 8 - Storage Features

The storage feature concerns retrieving data and storing user generated data if applicable. The product must either have local of remote storage.

Local

When using local storage all data is stored locally on the systems hard drive. Storage media can either be a flat file or a SQL Database.

Remote

When using remote storage all data is stored remotely. Storage can either be by means of a SQL Database or a WEB service.

Print Features

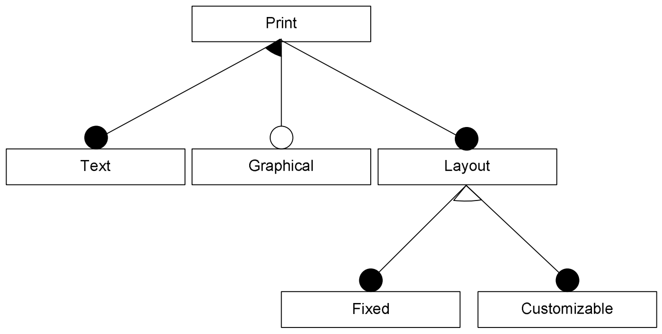


Figure 9 - Print Features

The print feature concerns printing of data (guides, item lists etc.). Printing must always support text print, and might optionally support graphical printing.

Layout

When printing a layout must exist. The layout can be fixed or customizable by the user.

Pricing Features

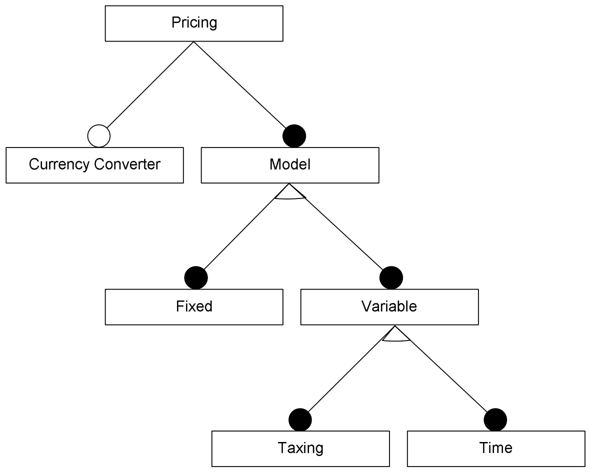


Figure 10 - Pricing Features

A product might optionally support pricing calculations. Pricing will not be applicable to all products, but might include a bill of materials calculation or a simple cheap, medium, expensive indication.

Pricing might have an optional currency converter and will always contain a model.

Model

The model prescribes how the price is calculates. The model might be fixed (like in the example above, or in simple summing of item prices), but might also be variable.

Variable model

The variable model might either be a taxing model, where specific taxing rules might apply to drinks depending on the type of liquor used and the country / state of use, specific taxing on spare parts etc. or a time model, where drink prices might be varied on week days, happy hour, etc.

Search Features

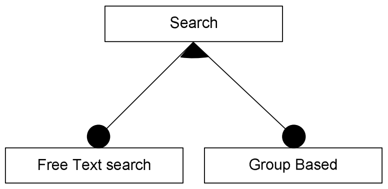


Figure 11 - Search Features

The search features concerns searching in data (guides, items etc.). Search can be free text search and group based (e.g. return all recipes using beef and chili with a medium price). Group based search is not applicable to all products.

5.2.3 Defining products

As a part of the workshop mentioned above (in a real scenario market analysis's etc. will most likely be part of this process), the following products are initially planned for the product line:

- DrinkMixer (Drink recipe product)
- Cookbook (Food recipe product)
- Engine Repair Guide
- Furniture Assembly Guide
- Cookbook Deluxe

The *Engine Repair Guide* might seem a bit odd, but in fact it's the same as the other recipe products, it is comprised of a set of needed ingredients, tools and an instruction. The search facility seems not to apply to this case ("I have two gaskets, one timing belt and four washers, what can I repair" doesn't make much sense). The requirement for illustrations accompanying the instruction would be important, but the other products will also benefit from this feature, so it is accepted as a core requirement.

An engine repair guide could benefit from a fault finding guide, e.g. “my car has these symptoms, which repair should I perform”. This concept is hard to transfer to the other planned products but future products might benefit from the functionality. This feature ends up in the “grey area” between features that are clearly in or out. For now it will not be part of the product line, but might be considered if customers require it or if future products might benefit from it.

The Cookbook deluxe was not planned from the start, but came up as an idea late in the process of discussing the other products. this is a good example of the continuous evolvement of a healthy product line.

5.2.4 Product line scope

To some degree it’s an open question if the product line is narrow- or wide scoped, as it can be discussed whether an Engine repair guide and a drink recipe product is of the same kind. In addition to this the product line features allow targeting for both home user and professional market segments (a drink mixer targeted for home use, will not have much use for taxing and happy hour pricing).

This and the fact that the GUI can be varied must classify the product line ad wide scoped.

5.2.5 Variations on planned products

The following table shows the specific combinations of features planned for the identified products.

Product	GUI	Guides	Storage	Printing	Pricing	Search
DrinkMixer	Desktop	Textual, unit and description	Local, flat file.	None	None	None
Cookbook	Desktop	Text and graphics. Units, description and user comments.	Local flat file.	Graphical print with pictures and text. Fixed layout.	Fixed, Level (low, med, high)	Free text and group based on ingredients.
Engine repair Guide	Web based	Text and graphics. Units, description, Additional info, user comments.	Remote, web service.	Graphical print with pictures. Fixed layout..	Variable, Taxing model.	Free text.
Furniture Assembly Guide	Desktop	Graphical with illustrations only. Units and description	Local, SQL database	Graphical, fixed layout	None	None
CookBook Deluxe	Desktop	Text and graphics. Units, description, user comments and ratings. (User comments and ratings can be shared between applications).	Remote, SQL Database.	Graphical print with pictures and text. Fixed layout.	Fixed, Level (low, med, high)	Free text and group based on ingredients, season, price group.

5.2.6 Quality Attribute Scenarios

Based on the feature modeling above a number of quality attribute scenarios have been identified concerning the modifiability of the software platform.

The different variations in products within the GUI, Storage, Pricing and Search features must be considered when designing the architecture. It must be possible to configure the system to support the various products and their variations.

In a product line where the software platform is used in various configurations, it is important that the variation in features can be handled easily without needing to perform a full test of the system as indicated in the Response and Response Measure of the quality attribute scenarios.

The following shows some of the QAS that will apply to the product line.

A developer changes the GUI from a web based GUI to a desktop GUI

Relevant Quality Attributes:		**Modifiability**
Scenario Details	Source:	A developer
	Stimulus:	Changes the GUI to a desktop based GUI
	Artifact:	The user interface
	Environment:	At build time
	Response:	Modification is made with no side effects
	Response Measure:	The modification is made within one working day
Questions:		
Issues:		The Response measure does not include the implementation time or customization of the web based GUI.

A developer changes the system to use an SQL database for storage instead of a flat file

Relevant Quality Attributes:		**Modifiability**
Scenario Details	Source:	A developer
	Stimulus:	Changes the system to use an SQL database
	Artifact:	The system
	Environment:	At build time
	Response:	Modification is made with no side effects
	Response Measure:	The modification is made within one working day
Questions:		Is it possible to change the print layout without modifying any code?

A developer changes the print layout

Relevant Quality Attributes:		**Modifiability**
Scenario Details	Source:	A developer
	Stimulus:	Changes the print layout
	Artifact:	The print layout configuration
	Environment:	Any time
	Response:	Modification is made with no side effects
	Response Measure:	The modification is made within one working day
Questions:		Is it possible to change the print layout without modifying any code?
Issues:		This quality may also be used to support user customizable print layouts at runtime.

A developer enables a pricing model for a product

Relevant Quality Attributes:		**Modifiability**
Scenario Details	Source:	A developer
	Stimulus:	Enables a pricing model
	Artifact:	The system
	Environment:	At build time
	Response:	Modification is made with no side effects
	Response Measure:	The modification is made within one working day
Questions:		
Issues:		The Response measure does not include the implementation time or customization of the pricing model.

A software product line will also have requirements for other qualities in the products. The following shows some example quality attribute scenarios that could apply to the product line:

User searches the database with maximum one second response times.

Relevant Quality Attributes:		**Performance**
Scenario Details	Source:	User
	Stimulus:	Searches the database
	Artifact:	System
	Environment:	During normal operation, up to 1000 users connected simultaneously.
	Response:	Searches are completed
	Response Measure:	The maximum latency is one second
Questions:		How many guides will the database contain?
Issues:		

Users browse the database with maximum one second response times.

Relevant Quality Attributes:		**Performance**
Scenario Details	Source:	User
	Stimulus:	Browses the database or a search result
	Artifact:	System
	Environment:	During normal operation, up to 1000 users connected simultaneously.
	Response:	The guide is presented on the GUI
	Response Measure:	The maximum latency is one second
Questions:		How much data will a guide contain, e.g. will there be any large images?
Issues:		

Unauthorized users are prohibited from reading or changing data from the system.

Relevant Quality Attributes:		**Security**
Scenario Details	Source:	An unauthorized user
	Stimulus:	Tries to read or modify data
	Artifact:	Guide data within the system
	Environment:	During normal operation in a multi-user system.
	Response:	The user is prohibited access from the system and an audit trail is maintained.
	Response Measure:	The incident is immediately logged in the security database.
Questions:		
Issues:		

An uneducated user learns to use DrinkMixer or CookBook within 20 minutes of usage.

Relevant Quality Attributes:		**Usability**
Scenario Details	Source:	Uneducated users
	Stimulus:	Uses the system for the first time
	Artifact:	DrinkMixer or CookBook
	Environment:	At runtime
	Response:	Users manages all functionality
	Response Measure:	Users have been working with the product for 20 minutes.
Questions:		
Issues:		

A new user of the EngineRepairGuide can be educated to use the system within one hour.

Relevant Quality Attributes:		**Usability**
Scenario Details	Source:	Uneducated users
	Stimulus:	Users are being educated by skilled users of the product
	Artifact:	EngineRepairGuide
	Environment:	At runtime
	Response:	Users manages all functionality
	Response Measure:	Users have educated for one hour.
Questions:		
Issues:		

5.2.7 Styles and Tactics

Performance

In order to achieve the wanted performance qualities of up to 1000 users searching or browsing the database with a maximum of 1 second response time we will

- Introduce concurrency

  Users must be able to search the database concurrently. Queries to the data storage will be processed asynchronously.

Modifiability

We will generally achieve modifiability in the system by

- Generalizing modules

  All modules will have well defined interfaces that support the wanted variation points.

- Program against interfaces

  To reduce the coupling between classes all code shall use interfaces. That way its easier to use classes with better algorithms at a later time. As a bonus this makes the system more testable.

- Database facade or a repository

  Many kinds of databases could be used for the system, ranging from flat text files to a DBMS or even a remote service. The details of what kind of database is used will be hidden by use of a facade or a repository.

- Where appropriate binding time will be deferred till runtime

  Optional components should be loaded at runtime, wherever this is practical and where other qualities don't suffer from this.

The print layout will partly be controlled by configuration. That is, it will be possible to configure which attributes and aggregates should be shown in the print but not the layout.

An intermediate format (e.g. in XML) will be introduced where also the layout is specified.

Usability

The system shall always allow the user to cancel operations and to undo her last operation.

Security

Users will have to be authenticated to use the system. All authenticated users will have full read and modify rights so no extra authorization is necessary.

To recover from unwanted incidents an audit trail will be kept, that can be used recover information and to exclude unwanted users.

5.3 Architecture Redesign

5.4 Architectural Prototype

6 Analysis and Results

7 Related work

[This section may be put in front of the hypothesis section or integrated into the method section, if it makes the flow of text more natural.]

In this section you outline what literature and other work your project build upon: papers, books, links to webpages, tutorials, etc. All references should be resolved in the reference section, that is do not use footnotes, or put the reference directly in the text. For an example, look how references are cited in

Bardram et al.[1].

It is good to address how your work extends, use, or build upon the cited work.

8 Conclusion

Your synopsis should clearly state: abstract, motivation, hypothesis, method, and (expected) analyses and results.

9 References

[1] http://sourceforge.net/projects/drinkmixer/

[[Bass et al., 2003] L. Bass, P. Clements, and R. Kazman. (2003) *Software Architecture in Practice 2nd Edition*. Addison-Wesley, chapter 14 and 15

[Clements et al., 2002] Clements, P. and Northrop, L. (2002) Software Product Lines.

[McGregor et al., 2002] McGregor, J.D., Northrop, L.M., Jarrad, S., Pohl, K. (2002) Initiating Software Product Lines. IEEE Software 19(4), pp 24-27

[Koschke, 2005] Koschke, R. (2005). What architects should know about reverse engineering and reengineering. WICSA Keynote, accessed 2008-05-07. http://www.informatik.uni-bremen.de/%7Ekoschke/koschke-keynote.pdf

[Deursen et al., 2004] van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C. (2004) Symphony: View-Driven Software Architecture Reconstruction. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), pp 122-132*

[Schmerl et al., 2006] Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H. (2006). Discovering Architectures from Running Systems. In IEEE Transactions on Software Engineering, vol 32, no. 7, pages 454-466

[Bardram et. al, 2004] Bardram, J., Christensen, H. B., and Hansen, K. M. (2004). *Architectural Prototyping: An Approach for Grounding Architectural Design and Learning.* In Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), pages 15–24, Oslo, Norway

[Floyd, 1984] C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer Verlag, 1984.

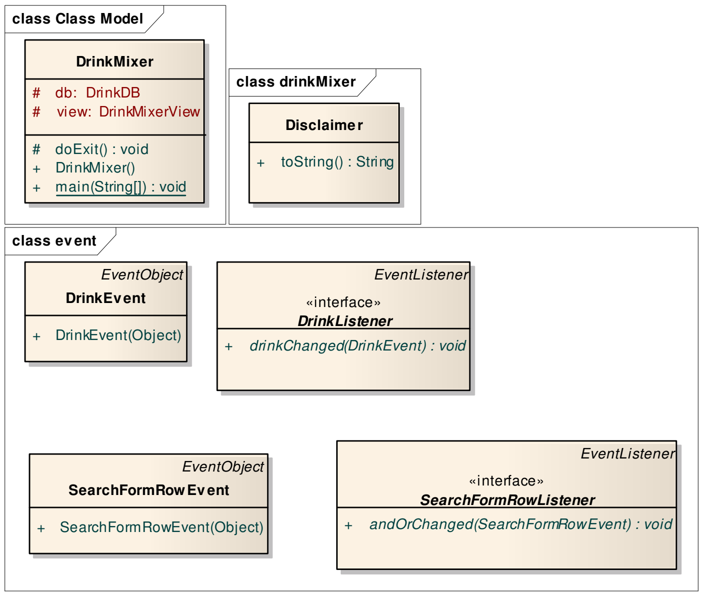
[Rational, 1998] Rational Software. Rational Unified Process: Best Practices for Software Development Teams. http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf, 1998.

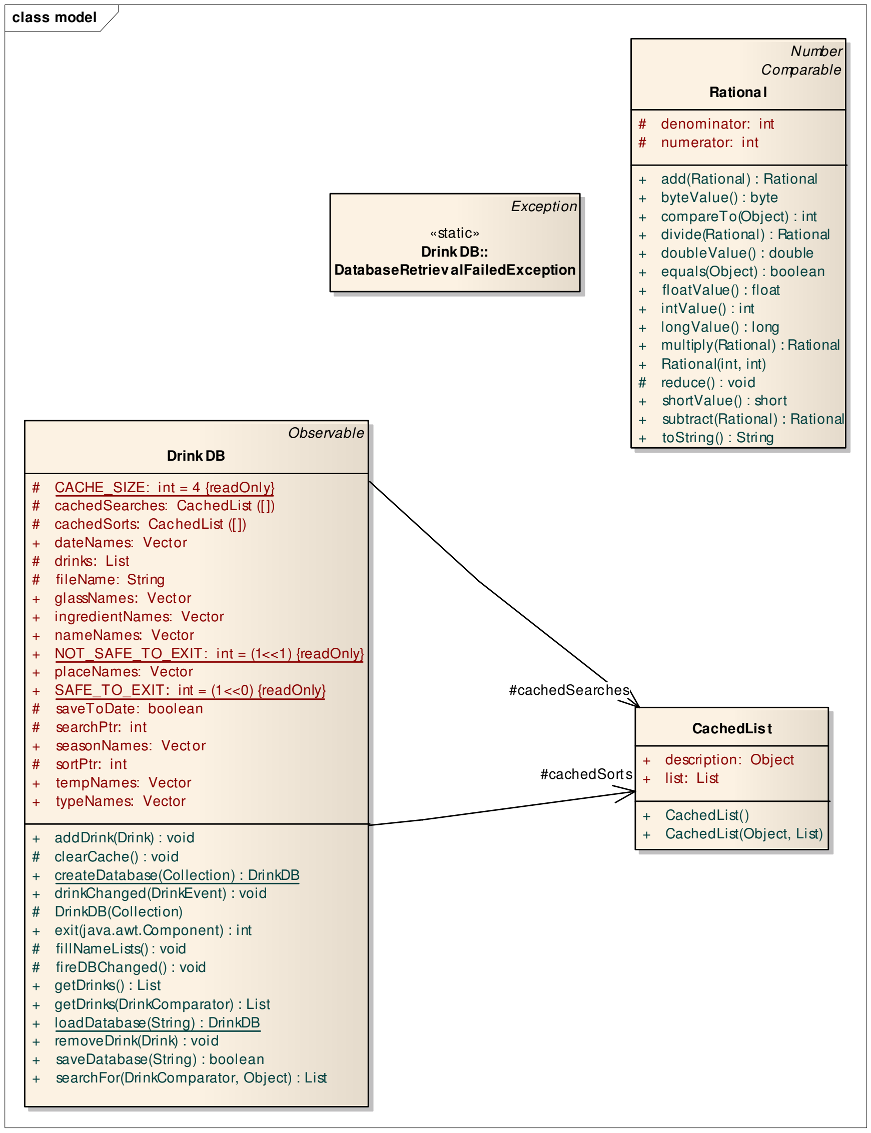
[Buschmann, 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) *Pattern – Oriented Software Architecture – A System of Patterns*. John Wiley & Sons

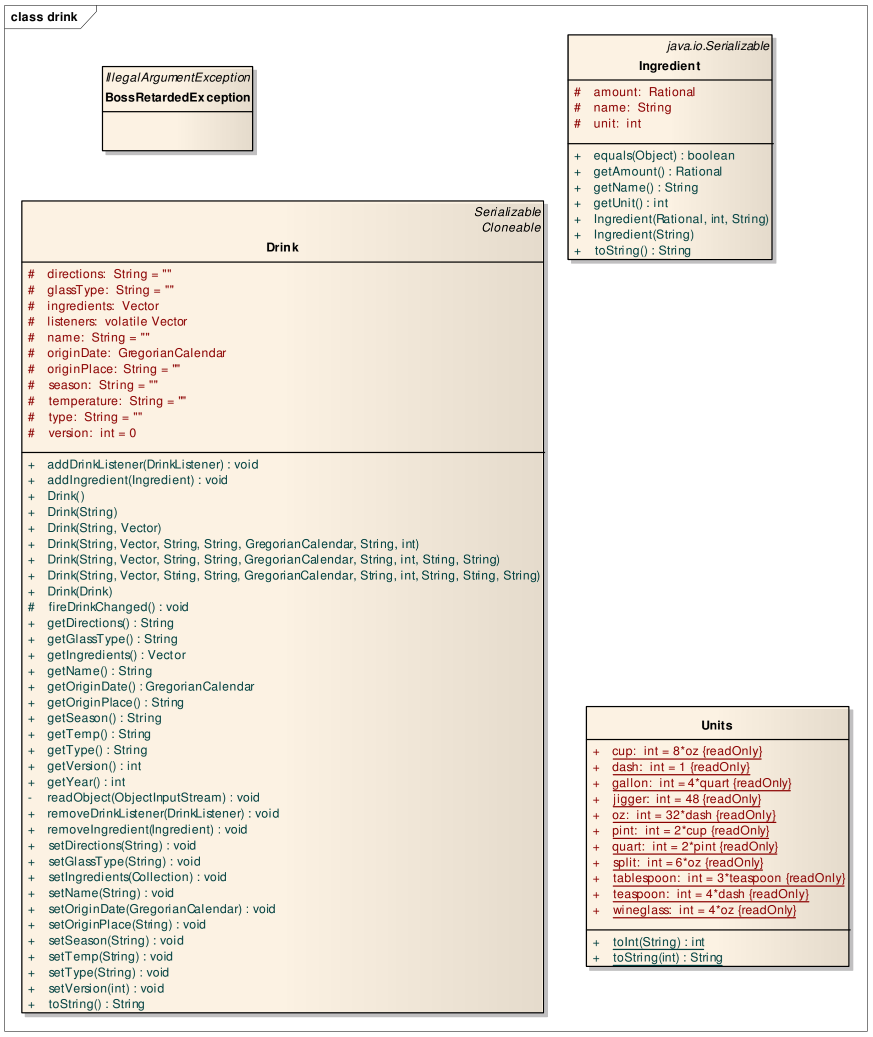
[Kang et al,1990] Kang, K. et al. (1990) *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute.

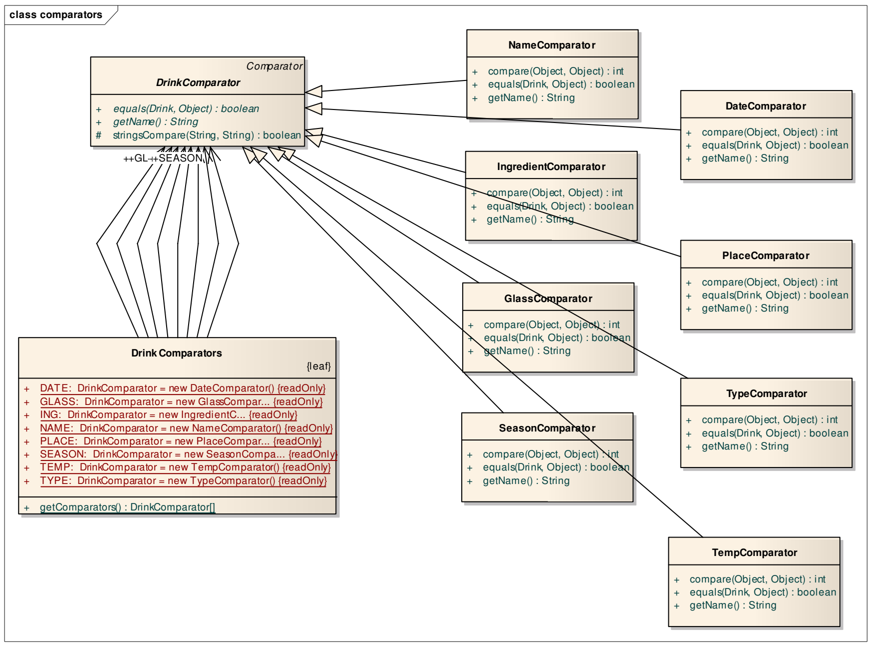
10 Appendix

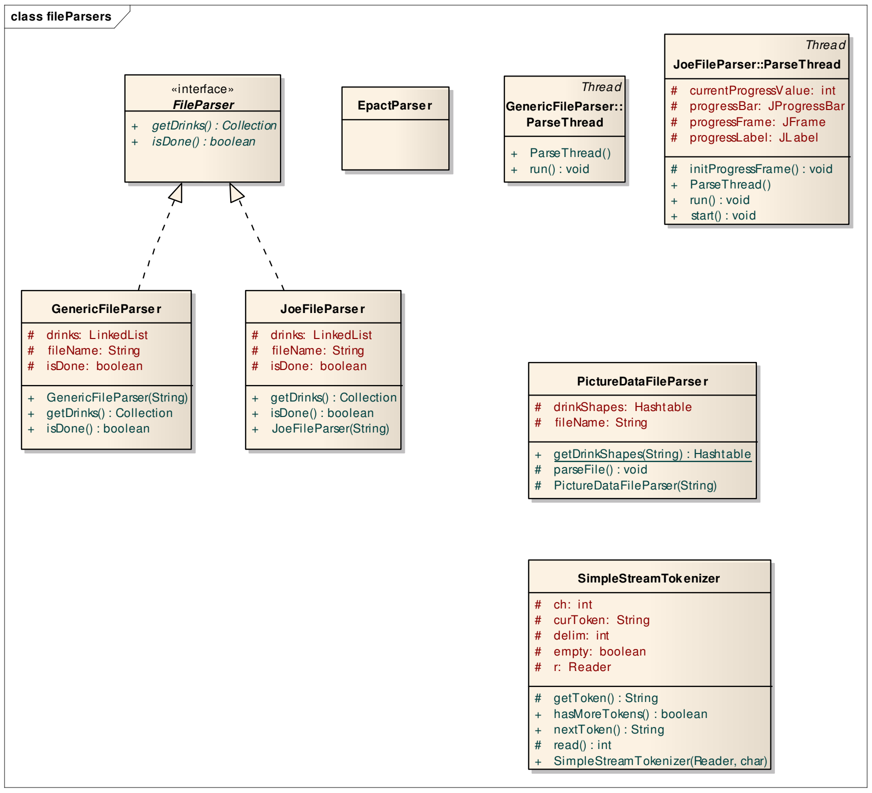
10.1 Autogenerated class diagrams

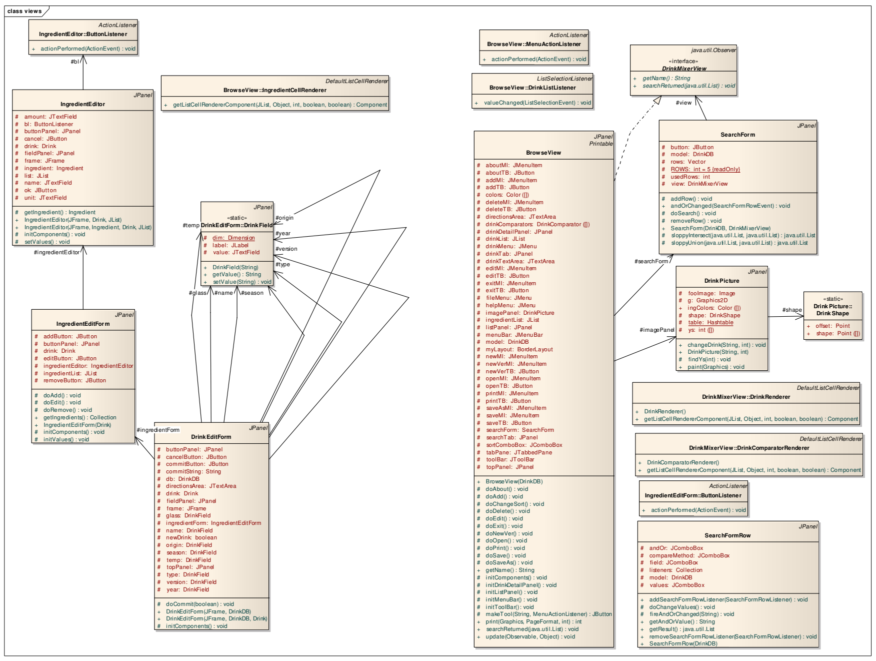



class model
Number
Comparable
Rational
# denominator: int
# numerator: int
+ add(Rational) : Rational
+ byteValue() : byte
+ compareTo(Object) : int
+ divide(Rational) : Rational
+ doubleValue() : double
+ equals(Object) : boolean
+ floatValue() : float
+ intValue() : int
+ longValue() : long
+ multiply(Rational) : Rational
+ Rational(int, int)
# reduce() : void
+ shortValue() : short
+ subtract(Rational) : Rational
+ toString() : String
Exception
«static»
DrinkDB::
DatabaseRetrievalFailedException
Observable
DrinkDB
# CACHE_SIZE: int = 4 {readOnly}
# cachedSearches: CachedList ([])
# cachedSorts: CachedList ([])
+ dateNames: Vector
# drinks: List
# fileName: String
+ glassNames: Vector
+ ingredientNames: Vector
+ nameNames: Vector
+ NOT_SAFE_TO_EXIT: int = (1<<1) {readOnly}
+ placeNames: Vector
+ SAFE_TO_EXIT: int = (1<<0) {readOnly}
# saveToDate: boolean
# searchPtr: int
+ seasonNames: Vector
# sortPtr: int
+ tempNames: Vector
+ typeNames: Vector
+ addDrink(Drink) : void
# clearCache() : void
+ createDatabase(Collection) : DrinkDB
+ drinkChanged(DrinkEvent) : void
# DrinkDB(Collection)
+ exit(java.awt.Component) : int
# fillNameLists() : void
# fireDBChanged() : void
+ getDrinks() : List
+ getDrinks(DrinkComparator) : List
+ loadDatabase(String) : DrinkDB
+ removeDrink(Drink) : void
+ saveDatabase(String) : boolean
+ searchFor(DrinkComparator, Object) : List
#cachedSearches
#cachedSorts
CachedList
+ description: Object
+ list: List
+ CachedList()
+ CachedList(Object, List)

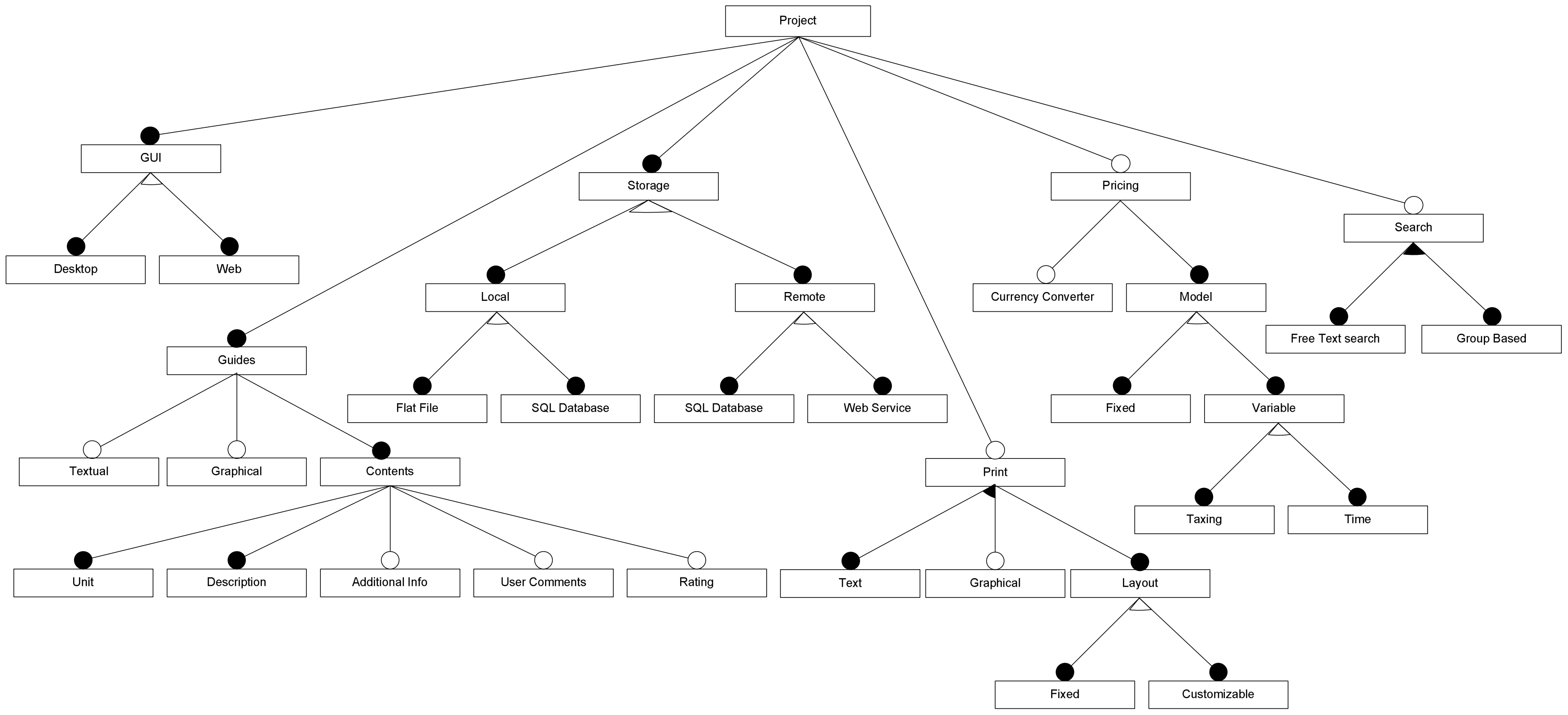

class drink
IllegalArgumentException
BossRetardedException
java.io.Serializable
Ingredient
# amount: Rational
# name: String
# unit: int
+ equals(Object) : boolean
+ getAmount() : Rational
+ getName() : String
+ getUnit() : int
+ Ingredient(Rational, int, String)
+ Ingredient(String)
+ toString() : String
Serializable
Cloneable
Drink
# directions: String = ""
# glassType: String = ""
# ingredients: Vector
# listeners: volatile Vector
# name: String = ""
# originDate: GregorianCalendar
# originPlace: String = ""
# season: String = ""
# temperature: String = ""
# type: String = ""
# version: int = 0
+ addDrinkListener(DrinkListener) : void
+ addIngredient(Ingredient) : void
+ Drink()
+ Drink(String)
+ Drink(String, Vector)
+ Drink(String, Vector, String, String, GregorianCalendar, String, int)
+ Drink(String, Vector, String, String, GregorianCalendar, String, int, String, String)
+ Drink(String, Vector, String, String, GregorianCalendar, String, int, String, String, String)
+ Drink(Drink)
# fireDrinkChanged() : void
+ getDirections() : String
+ getGlassType() : String
+ getIngredients() : Vector
+ getName() : String
+ getOriginDate() : GregorianCalendar
+ getOriginPlace() : String
+ getSeason() : String
+ getTemp() : String
+ getType() : String
+ getVersion() : int
+ getYear() : int
- readObject(ObjectInputStream) : void
+ removeDrinkListener(DrinkListener) : void
+ removeIngredient(Ingredient) : void
+ setDirections(String) : void
+ setGlassType(String) : void
+ setIngredients(Collection) : void
+ setName(String) : void
+ setOriginDate(GregorianCalendar) : void
+ setOriginPlace(String) : void
+ setSeason(String) : void
+ setTemp(String) : void
+ setType(String) : void
+ setVersion(int) : void
+ toString() : String
Units
+ cup: int = 8*oz {readOnly}
+ dash: int = 1 {readOnly}
+ gallon: int = 4*quart {readOnly}
+ jigger: int = 48 {readOnly}
+ oz: int = 32*dash {readOnly}
+ pint: int = 2*cup {readOnly}
+ quart: int = 2*pint {readOnly}
+ split: int = 6*oz {readOnly}
+ tablespoon: int = 3*teaspoon {readOnly}
+ teaspoon: int = 4*dash {readOnly}
+ wineglass: int = 4*oz {readOnly}
+ toInt(String) : int
+ toString(int) : String


class comparators
Comparator
DrinkComparator
+ equals(Drink, Object) : boolean
+ getName() : String
# stringsCompare(String, String) : boolean
++GL-++SEASON
NameComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
DateComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
IngredientComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
PlaceComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
GlassComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
TypeComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
SeasonComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
TempComparator
+ compare(Object, Object) : int
+ equals(Drink, Object) : boolean
+ getName() : String
DrinkComparators
{leaf}
+ DATE: DrinkComparator = new DateComparator() {readOnly}
+ GLASS: DrinkComparator = new GlassCompar... {readOnly}
+ ING: DrinkComparator = new IngredientC... {readOnly}
+ NAME: DrinkComparator = new NameComparator() {readOnly}
+ PLACE: DrinkComparator = new PlaceCompar... {readOnly}
+ SEASON: DrinkComparator = new SeasonCompa... {readOnly}
+ TEMP: DrinkComparator = new TempComparator() {readOnly}
+ TYPE: DrinkComparator = new TypeComparator() {readOnly}
+ getComparators() : DrinkComparator[]


class fileParsers
«interface»
FileParser
+ getDrinks() : Collection
+ isDone() : boolean
EpactParser
Thread
GenericFileParser::
ParseThread
+ ParseThread()
+ run() : void
Thread
JoeFileParser::ParseThread
# currentProgressValue: int
# progressBar: JProgressBar
# progressFrame: JFrame
# progressLabel: JLabel
# initProgressFrame() : void
+ ParseThread()
+ run() : void
+ start() : void
GenericFileParser
# drinks: LinkedList
# fileName: String
# isDone: boolean
+ GenericFileParser(String)
+ getDrinks() : Collection
+ isDone() : boolean
JoeFileParser
# drinks: LinkedList
# fileName: String
# isDone: boolean
+ getDrinks() : Collection
+ isDone() : boolean
+ JoeFileParser(String)
PictureDataFileParser
# drinkShapes: Hashtable
# fileName: String
+ getDrinkShapes(String) : Hashtable
# parseFile() : void
# PictureDataFileParser(String)
SimpleStreamTokenizer
# ch: int
# curToken: String
# delim: int
# empty: boolean
# r: Reader
# getToken() : String
+ hasMoreTokens() : boolean
+ nextToken() : String
# read() : int
+ SimpleStreamTokenizer(Reader, char)


class views
ActionListener
IngredientEditor::ButtonListener
+ actionPerformed(ActionEvent) : void
#bl
JPanel
IngredientEditor
# amount: JTextField
# bl: ButtonListener
# buttonPanel: JPanel
# cancel: JButton
# drink: Drink
# fieldPanel: JPanel
# frame: JFrame
# ingredient: Ingredient
# list: JList
# name: JTextField
# ok: JButton
# unit: JTextField
# getIngredient() : Ingredient
+ IngredientEditor(JFrame, Drink, JList)
+ IngredientEditor(JFrame, Ingredient, Drink, JList)
# initComponents() : void
# setValues() : void
#ingredientEditor
JPanel
IngredientEditForm
# addButton: JButton
# buttonPanel: JPanel
# drink: Drink
# editButton: JButton
# ingredientEditor: IngredientEditor
# ingredientList: JList
# removeButton: JButton
# doAdd() : void
# doEdit() : void
# doRemove() : void
+ getIngredients() : Collection
+ IngredientEditForm(Drink)
# initComponents() : void
# initValues() : void
#ingredientForm
DefaultListCellRenderer
BrowseView::IngredientCellRenderer
+ getListCellRendererComponent(JList, Object, int, boolean, boolean) : Component
JPanel
«static»
DrinkEditForm::DrinkField
# dim: Dimension
# label: JLabel
# value: JTextField
+ DrinkField(String)
+ getValue() : String
+ setValue(String) : void
#temp
#origin
#year
#version
#type
#glass
#name
#season
JPanel
DrinkEditForm
# buttonPanel: JPanel
# cancelButton: JButton
# commitButton: JButton
# commitString: String
# db: DrinkDB
# directionsArea: JTextArea
# drink: Drink
# fieldPanel: JPanel
# frame: JFrame
# glass: DrinkField
# ingredientForm: IngredientEditForm
# name: DrinkField
# newDrink: boolean
# origin: DrinkField
# season: DrinkField
# temp: DrinkField
# topPanel: JPanel
# type: DrinkField
# version: DrinkField
# year: DrinkField
# doCommit(boolean) : void
+ DrinkEditForm(JFrame, DrinkDB)
+ DrinkEditForm(JFrame, DrinkDB, Drink)
# initComponents() : void
ActionListener
BrowseView::MenuActionListener
+ actionPerformed(ActionEvent) : void
ListSelectionListener
BrowseView::DrinkListListener
+ valueChanged(ListSelectionEvent) : void
JPanel
Printable
BrowseView
# aboutMI: JMenuItem
# aboutTB: JButton
# addMI: JMenuItem
# addTB: JButton
# colors: Color ([])
# deleteMI: JMenuItem
# deleteTB: JButton
# directionsArea: JTextArea
# drinkComparators: DrinkComparator ([])
# drinkDetailPanel: JPanel
# drinkList: JList
# drinkMenu: JMenu
# drinkTab: JPanel
# drinkTextArea: JTextArea
# editMI: JMenuItem
# editTB: JButton
# exitMI: JMenuItem
# exitTB: JButton
# fileMenu: JMenu
# helpMenu: JMenu
# imagePanel: DrinkPicture
# ingredientList: JList
# listPanel: JPanel
# menuBar: JMenuBar
# model: DrinkDB
# myLayout: BorderLayout
# newMI: JMenuItem
# newVerMI: JMenuItem
# newVerTB: JButton
# openMI: JMenuItem
# openTB: JButton
# printMI: JMenuItem
# printTB: JButton
# saveAsMI: JMenuItem
# saveMI: JMenuItem
# saveTB: JButton
# searchForm: SearchForm
# searchTab: JPanel
# sortComboBox: JComboBox
# tabPane: JTabbedPane
# toolBar: JToolBar
# topPanel: JPanel
+ BrowseView(DrinkDB)
# doAbout() : void
# doAdd() : void
# doChangeSort() : void
# doDelete() : void
# doEdit() : void
# doExit() : void
# doNewVer() : void
# doOpen() : void
# doPrint() : void
# doSave() : void
# doSaveAs() : void
+ getName() : String
# initComponents() : void
# initDrinkDetailPanel() : void
# initListPanel() : void
# initMenuBar() : void
# initToolBar() : void
# makeTool(String, MenuActionListener) : JButton
+ print(Graphics, PageFormat, int) : int
+ searchReturned(java.util.List) : void
+ update(Observable, Object) : void
java.util.Observer
«interface»
DrinkMixerView
+ getName() : String
+ searchReturned(java.util.List) : void
#view
JPanel
SearchForm
# button: JButton
# model: DrinkDB
# rows: Vector
# ROWS: int = 5 {readOnly}
# usedRows: int
# view: DrinkMixerView
# addRow() : void
+ andOrChanged(SearchFormRowEvent) : void
# doSearch() : void
# removeRow() : void
+ SearchForm(DrinkDB, DrinkMixerView)
# sloppyIntersect(java.util.List, java.util.List) : java.util.List
# sloppyUnion(java.util.List, java.util.List) : java.util.List
#searchForm
#imagePanel
JPanel
DrinkPicture
# fooImage: Image
# g: Graphics2D
+ ingColors: Color ([])
# shape: DrinkShape
# table: Hashtable
# ys: int ([])
+ changeDrink(String, int) : void
+ DrinkPicture(String, int)
# findYs(int) : void
+ paint(Graphics) : void
#shape
«static»
DrinkPicture::DrinkShape
+ offset: Point
+ shape: Point ([])
DefaultListCellRenderer
DrinkMixerView::DrinkRenderer
+ DrinkRenderer()
+ getListCellRendererComponent(JList, Object, int, boolean, boolean) : Component
DefaultListCellRenderer
DrinkMixerView::DrinkComparatorRenderer
+ DrinkComparatorRenderer()
+ getListCellRendererComponent(JList, Object, int, boolean, boolean) : Component
ActionListener
IngredientEditForm::ButtonListener
+ actionPerformed(ActionEvent) : void
JPanel
SearchFormRow
# andOr: JComboBox
# compareMethod: JComboBox
# field: JComboBox
# listeners: Collection
# model: DrinkDB
# values: JComboBox
+ addSearchFormRowListener(SearchFormRowListener) : void
# doChangeValues() : void
# fireAndOrChanged(String) : void
+ getAndOrValue() : String
+ getResult() : java.util.List
+ removeSearchFormRowListener(SearchFormRowListener) : void
+ SearchFormRow(DrinkDB)

10.2 Feature Model


Project
GUI
Desktop
Web
Storage
Local
Flat File
SQL Database
Remote
SQL Database
Web Service
Pricing
Currency Converter
Model
Fixed
Variable
Taxing
Time
Search
Free Text search
Group Based
Guides
Textual
Graphical
Contents
Unit
Description
Additional Info
User Comments
Rating
Print
Text
Graphical
Layout
Fixed
Customizable