

2 Foundations

A little knowledge that acts is
worth infinitely more than
much knowledge that is idle.

(Khalil Gibran)

Contents

2.1	Metrics	5
2.2	Variability	13
2.3	Enterprise JavaBeans (EJB) 3.0	21
2.4	The MeDIC Information System	23

The central topic of this master thesis is to enhance the meta-model of the MeDIC information system. The MeDIC itself is a metric management system; therefore the basic knowledge about metrics and process specification of the metrics are needed to understand the enhancement proposed in this works. Furthermore, the fundamental concepts to support the analysis about meta-modeling and variability are covered. Moreover, the technologies that are used in the current implementation of the MeDIC system are introduced (Enterprise JavaBeans 3.0, Web Services, etc). The content is not intended to be comprehensive.

2.1 Metrics

Measurement is essential for understanding, defining, managing and controlling software development and maintenance processes. Metrics lies on the needed of measurement, because we cannot control something that we cannot measure. Metrics provide a quantitative basis to evaluate the changes in software process. One of the main reasons of the growing interest in software metric has been the perception that software metrics are necessary for software improvement.

2.1.1 Metric Introduction

Some basic terms are used in the metric area; metric itself, measure, and indicator. It is important to understand the differences between these terms, because in some practices the definition and terminology of these terms seems to be confusing and interchanging. Therefore, the metric introduction will be started

with the definitions of those terms.

Fenton and Pfleeger [FP97] define metric as:

"The process by which number or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to the clearly defined rules"

An entity is an object such as a software module, while an attribute is a measurable property of the object. Entities fall into three categories; products, processes and resources (Fenton and Pfleeger). A process is any activity related to software development, a product is any artifact produced during software development and a resource is people, hardware, or software needed for the processes (Fenton and Pfleeger).

McGarry [McG01] defines a measure as:

"A variable to which a value is assigned to represent one or more attributes"

Indicator is defined as:

"A device or variable that can be set to a prescribed state based on the result of a process or the occurrence of a specified condition." [IEEE, 1990]

Those terms can be viewed as a hierarchy with indicator at the top and measure in the bottom, shown in the figure 2.1 below. A metric is an expression composed of one or more measures. For example, a metric to find the number of defects during testing process. The measure can be defect count or phase defect detected, while the indicator is the reduction number of the defects found in testing. Indicators convey the significance of the metrics in the specified environment.

There are two general types of the metrics; direct and indirect measurement.

Direct measurement Direct measurement of an entity attribute involves no other attribute or entity. For example, we can measure the length of a physical object without any other object. Measures below are example of the direct measures used in software engineering provided by Fenton and Pfleeger:

- Length of code (LOC)
- Duration of testing process (hours)
- Number of defect discovered during testing process (number of de-

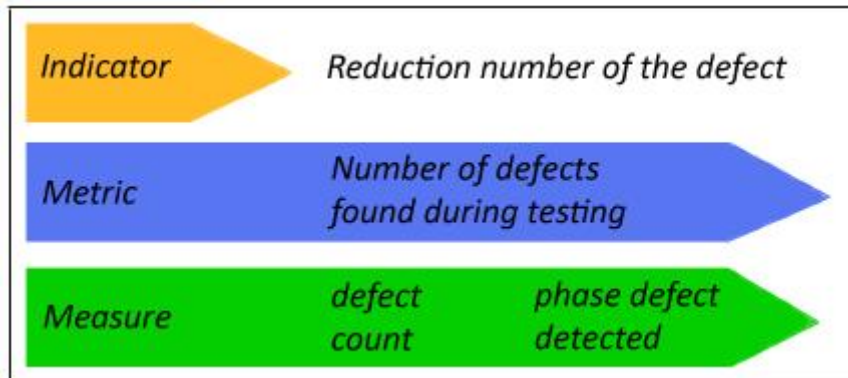


Figure 2.1: Hierarchy Of Measurement Terms

fect)

- Time a programmer spent on a project (months)

Indirect measurement Indirect measurements are measures of an attribute obtained by comparing different measurements. The difference between direct and indirect measurement is the metric function, one variable in direct measurement and n-tuple in indirect measurement. Some common examples of derived measurement in software engineering are:

- Programmer productivity (code size/programming time)
- Module defect density (number of defects/module size)

In the metric term, direct measurement will be referred as a base metric and indirect measurement as a derived metric. A derived metric contain two or more base/derived metrics. A value of a measure can be collected directly from base metric, while the value from derived metric is a result of function calculation from one or more base/derived metrics.

2.1.2 Metric Process

According to Fenton and Pfleeger, the process of defining new metrics involves three steps: identify measurement entities, identify attributes of the entity that are to be measured, and then define new metrics that can be used to measure each attribute. These steps should be executed in the order in which they appear.

Goal-Quality-Metric (GQM)

Goal-Quality-Metric is a top-down approach to establish a goal-driven measurement system for software development, one of the methods for creating metrics to meet specific information need. This approach was developed by Basili et al

[BCR94] at the Software Engineering Laboratory (SEL), NASA Goddard Space Flight Center during 1980, was refined during 1990, and now, serves as the foundation framework for many measurement initiatives.

The GQM approach divides the process into three levels; conceptual, operational, and quantitative level. Goals are identified in conceptual level; describing what general objective wants to be achieved. On operational level, questions that help to understand how to meet the goal are formulated. Metrics identify the measurements that are needed to answer the question. They are defined on the quantitative level. The GQM approach is illustrated in the figure below.

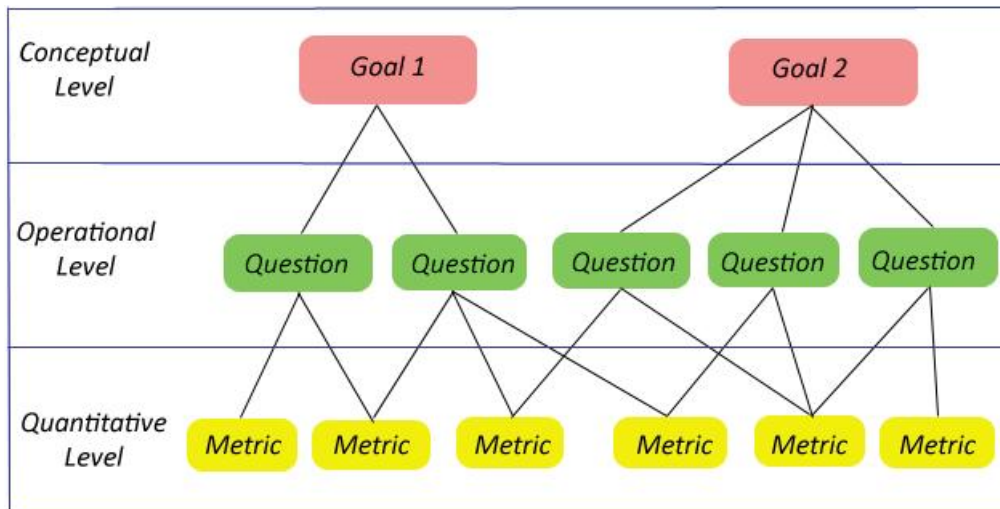


Figure 2.2: Goal-Question-Metric approach

GQM approach is divided into three levels as explained before. Goals identify what we want to accomplish; questions, when answered, tell us whether we are meeting the goals or help us understand how to interpret them; and the metrics identify the measurements that are needed to answer the questions and quantify the goal. The mapping between goals, questions, and metrics is not a one-to-one relationship. A single measurement goal may apply to multiple business goals and vice versa; for each goal, there can be several questions and the same question can be linked to multiple goals as appropriate. For each question, there can be multiple metrics, and some metrics may be applicable to more than one question. This hierarchical structure helps ensure that the measurement program focuses on the right metrics and that we avoid extra work associated with collecting metrics that are not really needed.

[BCR94] Basili described six-step GQM process as follows:

1. Establishing Goals
 - Develop a set of corporate, division and project business goals and associated measurement goals for productivity and quality
2. Generating Questions

Generate questions (based on models) that define those goals as completely as possible in a quantifiable way

3. Specifying the Measures
Specify the measures needed to be collected to answer those questions and track process and product conformance to the goals
4. Specifying the Measures
Specify the measures needed to be collected to answer those questions and track process and product conformance to the goals
5. Preparing for Data Collection
Develop mechanisms for data collection
6. Collecting, Validating and Analyzing the data for Decision Making
Collect, validate and analyze the data in real time to provide feedback to projects for corrective action
7. Analyzing the Data for Goal Attainment and Learning
Analyze the data in a postmortem fashion to assess conformance to the goals and to make recommendations for future improvements

One of the key practices of the GQM approach is to derive appropriate metrics. For a given question, there are many relevant metrics. The key is to identify or choose those metrics that clearly satisfy the question. Creating metrics more than are really needed cause extra work and cost. GQM ensures that each metric has purpose, and no metrics are defined without a purpose. The advantages of the GQM approach are listed as follows [HRY10]:

- Ensure the adequacy, consistency and integrity of metrics plan and data collection. The designer of metrics program (that is metrics analyst) must get a lot of information and the dependence between them. To ensure the metrics collection is adequate, consistent and integrated, the analyst should understand why to metrics these properties accurately, what is the underlying assumption, and what the model will be applied to the use of metrics data.
- Help to manage the complexity of metrics plan. When a large number of measurable attributes exist and the number of metrics for the attributes increases accordingly, the degree of complexity of the metrics plan will undoubtedly increase. In addition, the approach selected in order to adequately metrics an attribute also depends on the goal of metrics. If you do not have a goal-driven framework, the metrics plan will soon be out of control. No one mechanism capturing the dependence between attributes, the metrics plan is very easy to introduce inconsistency to any changes.
- Help organizations to discuss the metrics and improvements of the goal driven on the structure of the common understanding and eventually from

a consensus. In turn, this also enabled the organizations to define the metrics and models accepted widely in the organizations.

Goal-Attribute-Measure (GAM)

Goal Attribute Measure (GAM) proposed by Sazama [Fra08] is a modification method of the aforementioned GQM method that focuses on attribute definitions rather than question formulation. Sazama described two issues of formulating question in GQM; the questions are frequently too close to the target definition or too close to the metrics. To derive measures in GAM, firstly we have to identify measurement customers and their goals. Then a set of target attributes is determined. In the next step, measured attributes are assessed to find metric that can measure the property directly. If there is no such metric is found, the attribute must be broken down into sub-attributes until all the attributes can be measured with the metrics. In GAM, the scope of goals is on measurement objects while focus is on the structuring and definition of attributes.

Approach	Architecture	Scope	Focus
GQM	Goal Question Metric Question Metric	Project	Question Definition
GAM	Goal Attribute Measure	Measurement object	Attribute structuring & definition

Table 2.1: GQM and GAM comparison table

From the whole explanation, there is only small difference between GQM and GAM. But from the table above, we can conclude that GAM is best used to measure specific objects; and GQM is used when measuring a software project as a whole.

2.1.3 Metrics in the Organization

Software metrics as quantitative standards of measurement for various aspects of software projects being used by many software organizations in order to control software project, process, and products. Furthermore, a well designed metric will support decision making by management and enhance the return of investment in the organizations. In traditional way, the measurement process was a trivial task that does not get enough attention from management. As the growth of software engineering knowledge to control and improve the quality of process, measurement became one of the key points of process improvement standard in Capability Maturity Model Integration (CMMI) [SEI10].

The key process area of CMMI that covers the use of metrics is Measurement and Analysis (MA), which specifies that software organization must be able

to perform measurement with focus on project management in order to reach maturity level 2. The purpose of this process area is to develop and sustain a measurement capability that support management information needs. In more detail, the Measurement and Analysis process area involves:

- Specifying the objectives of measurement and analysis such that they are aligned with identified information needs and objectives
- Specifying the measures, data collection and storage mechanisms, analysis techniques, and reporting and feedback mechanisms
- Implementing the collection, storage, analysis and reporting of the data
- Providing objective results that can be used in making informed decisions and taking appropriate corrective actions

Organizational wide defined metrics are applied throughout the organization in management level to achieve the main goal of the organization. Therefore, the specific usage of metrics is mostly placed in projects within the organization. Metrics that are defined in a specific project are called as project specific metrics. Along with projects as a part of the organization, all defined project specific metrics must be derived from organizational wide defined metrics. Each project has different information needs. Therefore, project specific metrics derived from organizational wide defined metrics need to be adjusted and adapted to meet specific project condition. Based on Tavizon's works [Tav11], the changes of metrics are called variations of the organizational wide defined metrics, while the adjustment process is called tailoring.

On project level of the organization, metrics can provide key indicators of project achievements, adherence, and quality. On management level, McGarry lists five area that can be enhanced with metric data:

- **Effective Communication**
Metrics help the decision maker to manage business goals and associated tasks at all levels within the organization and communicate the health of the organization.
- **Track Project Plan Goals**
Metrics help to describe the status of the project regarding its processes and products. Metrics also represent the progress of the activities being executed and the quality of their results.
- **Risk Management**
Metrics assist with a proactive management strategy. For example, estimations can be analyzed and potential problems could be better evaluated and prioritized. It is known that the earlier a problem is discovered, the less cost it will represent for the organization and less problematic to solve it.

- **Elaborate Key Trade-off Decisions**
The projects are subject to constraints and decisions being made in one area, which certainly would impact another different area. These impacts need to be assessed. The results from the assessment can be used to elaborate trade-offs meeting the project goals.
- **Rationalize Decisions**
Decision makers, technical and project managers, must be able to defend their estimates and plans with historical data. Metrics provide a solid rationale for selecting the best available option.

It is important to notice, metric data by itself does not guarantee that a project will succeed. Nevertheless, it provides the decision makers with the sufficient information to deal with critical and non-critical issues inherent in projects and to follow a proactive approach. For this reason, metric data supports the projects and consequently the organization, to succeed. However, it must be pointed out that metrics are only useful for bigger projects, in small project the cost in development, interpretation, and calculation of the metrics might too big to cover.

In the practical usage of metrics in an organization, metrics are selected from several existing common used metrics to support the business goals. These metrics are highlight metrics in software development management:

- Source code growth rate reflects requirements completeness and the software development process.
- Effort data reflect the nature of the project environment and the type of problem being solved.
- System size estimates reflect requirements stability and completeness within the environment.
- Computer usage, which is directly related to the particular process being applied.
- Error rates reflect the total number of errors vs. estimated errors.
- Reported/corrected software discrepancies allow gaining insight into software reliability, progress in attaining test completion, staffing weaknesses and testing quality.

Choosing metric based on business goals and implementing them properly can have measureable impact on the business result. But from all existing standard metrics, the implementation of metrics might not fully achieve the project goal. Therefore, adjustment metrics based of each project purposes or defining new metrics are recommended to get better result of metric usage.

2.2 Variability

Variability modeling is a domain specific modeling technique that helps managing complexity and facilitates reuse, with feature decomposition. Variability modeling was first introduced for managing variability in software product families, exploiting the similarities within a set of products to reduce the product development cost. By now, variability became a central concept in software product line engineering. Several variability techniques have been developed to address variability based on each engineering disciplines, include software engineering. Variability modeling consists of variation points, variants, and relationships between them. A variation point is a representation of a variable item of the real world or a variable property of such an item. Based on the visibility of the variation point to users and experts, variation points are classified in two types, the external or internal variation point. Another classification of variation points is based on the possibility or restriction on providing or selecting a variant, which are either opened or closed. Variants in opened variation point are provided by users, while variants of closed variation points are provided by the system, users are not allowed to add more variants but select variant from an available list .

A variant is a representation of a particular instance of a variation point, therefore, a variant should have a relationship with at least one variation point [BB07]. The variants can be categorized into two types; primitive and complex. A primitive variant is represented by a number or character, while other kind of variants will be categorized as complex variant, such as an interval type of variant.

Relationship between variation points and/or variants is distinguished into variability dependency and constraint dependency. Variability dependency shows a variability relationship between variation points and its variants: mandatory, optional or alternative dependency. Variant with mandatory dependency have to be chosen if its variation point is selected. An optional dependency indicates that none, one or more variants can be selected. Alternative dependency declares the range between a variation point and its variant, whether the option of none or one variant (or_alternative) or option of one or more variants (xor_alternative) for each variation point. The constraint dependency distinguishes a requires and an excludes dependency between variants, variation points, and variation points to variants. A requires constraint dependency shows that a variability is depend on another variability, on the other hand, an excludes constraint dependency indicates that a variability has to eliminate another variability.

2.2.1 Source of Variability

In more detail, there are some sources of variation in architecture design process of software engineering [BL01]:

Variation in function A particular function may exist in some products and not in others. For example, consider a car radio/navigation system within an automobile. Some automobiles may have a radio and no navigation, others navigation without the radio and still others may have both. The characteristics of the radio will vary across different products as well. This situation may also arise within a single product if the requirements are not known as the design proceeds.

Variation in data A particular data structure may vary from one product to another. For example, assume in a call center application two components exchange information about a customer. This information contains among other things the mailing address, which is realized as an unstructured text string. To support a feature in another version of the call center application (e.g. a structured display of the customer's mailing address) the format of the mailing address has to be different. Variation in data in most cases is a consequence of variation in function.

Variation in control flow A particular pattern of interaction may vary from one product to another. For example, assume there is a notification mechanism between components in place that informs interested components that some data values have been changed. One possible implementation is that all the components get notified in sequence within a single control flow. In a particular product some of the components to be notified may actually be 3rd party components, which may have some unknown behavior. For reliability reasons it might be a good idea to direct the control to a component that is able to do an error recovery in case a notified component does not return the control.

Variation in technology The platform (OS, hardware, dependence on middleware, user interface, run-time system for programming language) may vary in exactly the same fashion as the function. A particular piece of middleware may be required in one product and not in another. The OS or the hardware may vary from product to product. For example, a sensor may be connected directly to the controller whose software is being designed or it may be connected over a communication line. If the sensor is connected directly to the controller, then sensor management software is needed, if it is connected over a communication line then communication line management software is needed.

Variation in quality goals The particular quality goals important for a product may vary. For example, the coupling between a producer and consumer of a data item may be achieved via a publish subscribe mechanism or via a

permanent connection. The choice of one or another of these two options embodies a choice of the importance of performance and modifiability and this choice may be different in different products.

Variation in environment The fashion in which a product interacts with its environment may vary. For example, a particular piece of middleware may be invoked from either C++ or Java. The invocation mechanism may vary from one product to another.

2.2.2 General Variability Meta-Model

In order to reduce the complexity and gain understanding about variability, the variability meta-model as a centralized modeling of variability proposed by Bühne et al [BLP04] provided in figure 2.3. This variability meta-model expressed all the variability components that have been described in section 2.2.1. . Variation point and variant are the main components in variability modeling. The variability dependency shows the relation between variation point and variant and specializes into three: alternative, optional, and mandatory. The relation between variation points, variants, and between variant and variation point to constraints the variability model represent with constrain dependency, whether it is requires or excludes.

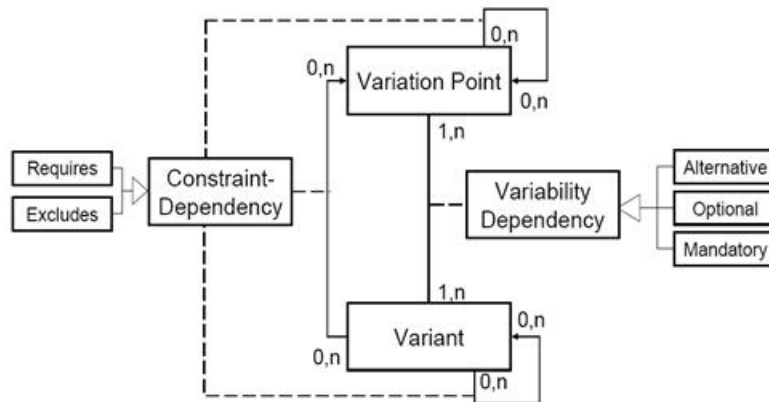


Figure 2.3: The variability meta-model

2.2.3 Variability Mechanism

Modeling variability is related with variability mechanisms and variability techniques. Variability mechanisms are common ways to introduce or implement variability. While variability techniques model the variability that is provided by the product artifacts with well defined language for representing the variability [GW04]. Variability techniques are aimed to support variability management during product derivation. Several variability techniques have been developed

with their own concepts to capture variability, some of the techniques are supported with tools to model variability. This section will focus on variability mechanisms.

Gomaa and Webber [GW04] describe four different mechanisms to model the variability; using parameterization, information hiding, inheritance, and variation points. Furthermore, other mechanisms have been indentified, such as conditional compilation, patterns, generative programming, macro programming, and aspect oriented programming. Not all mechanisms will be discussed here, but four mechanisms based on Gomaa and Webber as a basic variability mechanisms and pattern mechanism that will be discussed further in this master thesis.

Parameterization

Parameterization is variability mechanism where the variation is to the value of the parameters, which allows the user to change the values of attributes by providing the capability through the component's interface to initialize or change the value of parameterized attributes. Modeling variability with parameterization takes shorter time for development, but the variability is limited as no functionality can be changed.

Information Hiding

Modeling variability using information hiding is where several version of the same component are built with the same interface. The variants are the different versions of the same component, and the variability is hidden inside each version. The user selects a component from an available set of choices and inserts it into the application. Using information hiding, the variability is limited to the available choices, but this mechanism offers higher variability than parameterization because the functionality can be varied.

Inheritance

The inheritance variability mechanism is shown as generalization-specification hierarchy, where the variants do not need to adhere to the same interface. Variants is specialization of others components, there subclasses can extend the interface of superclass by adding new operations or override existing operations.

Variation Point

Modeling variability using variation points allows the user to create specific and unique variants. Using variation points, the user may build a system component with unique variants built from the variation points. Modeling variability using variation points provides the most variability and flexibility in creating an application.

Pattern

A design patterns is a proven design solution for a particular problem that has been used in many applications. Keepence and Mike [KM99] applied the pattern mechanism to model variability of spacecraft mission-planning system. The pattern applied in the spacecraft mission-planning system is adapter pattern,

which is deal with variability in structural rather than behavior functionality. Patterns provide reusable, routine solutions to certain types of problems and support the reuse of underlying implementations. Modeling variability using patterns starts by analyzing the user requirements from systems and build an object-oriented family model using a set of predefined patterns. With pattern, mapping from requirements to implementation is directly visible. Pattern permits user to model variants using a single pattern, and support modeling for complex system. Another design pattern that are frequently referred to variability mechanisms are Strategy, Template Method, Factory, Decorator and Builder pattern.

In the implementation of variability more than one variability mechanism can be combined to find the best way in introducing variability to the system. Several variability mechanisms have been developed from the basic mechanisms above to adjust particular situation.

2.2.4 Variability Techniques

Over the past years, a lot of research regarding variability have been conducted with several publication and modeling techniques as the results. These variability modeling techniques aim at representing and managing the variability more easily. Each of these variability techniques has similarities and differences among the others in modeling variability and tools support, which is designed for a particular situation. Feature modeling with feature diagram was first introduced in Feature Oriented Domain Analysis (FODA) by Kang et al [KCH+90]. Feature modeling has been widely adopted for modeling variability in software product line, afterwards, several variability modeling techniques and tool supports were developed based on the initial approach presented in the feature modeling.


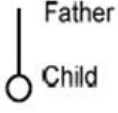
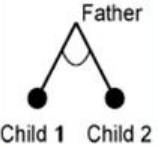
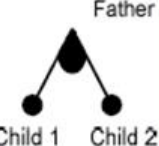
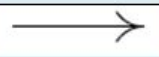

Feature Modeling

The aim of feature modeling was to visualize the variability domain in feature diagram. The primary focus was to establish commonalities and differences of a product as features. These features are the basic building block of any feature model. The definition of feature based on Kang et al:

"A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"

Feature models express the variability through mandatory, optional, and alternative features, and dependencies between features. The main notations of feature diagram are represented in the table below:

Table 2.2: Notation of feature diagram

Type	Semantic	Characteristic	Notation
Domain Relationship			
Mandatory	If the father feature is selected, the child feature must be selected as well		
Optional	If the father feature is selected, the child feature can but need not to be selected		
Alternative	If the father feature is selected, exactly one feature of the alternative-child-features must be selected	Implicit mutually exclusion between alternative-child-features	
Or	If the father of feature is selected, at least one feature of the or-child features must be selected		
Dependency Relationship			
Implication	If one feature is selected the implied feature has to be selected as well, ignoring their position in the feature tree	Transitive	
Exclusion	Indicates that both features cannot be selected in one product configuration and are therefore mutually exclusive	Symmetric	

A feature diagram can be defined as a visualization of notation that hierarchically structures the set of features from the model, which is basically a tree. A simple car prototype example presented in figure 2.4. The car prototype diagram specifies possible configuration for a car, which has mandatory parts (cars body, transmission and engine) and optional car feature (pulls trailer). The transmission can exclusively be automatic or manual, while the engine can be an electric engine and/or gasoline.

Feature modeling has been more and more appreciated by requirements engineer. Therefore, the quality of a feature model is determined by a given domain and integrity of the model itself. As the adequate capture domain can hardly be analyzed and reviewed, the integrity of the model can shows deficiencies as mention by Maßen and Lichter below:

- Redundancy , A feature model contains redundancy, if at least one semantic information is modeled in multiple ways
- Anomaly, A feature model contains anomalies, if potential configurations are being lost, though these configurations should be possible.

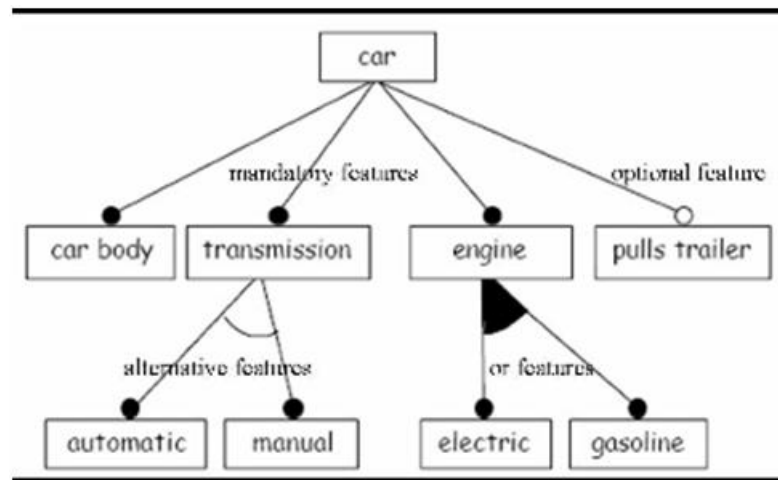


Figure 2.4: Feature diagram example - Car prototype diagram

- Inconsistency, A feature model contains inconsistencies, if the model includes contradictory information.

Others Variability Techniques

Other variability techniques have been developed as extension of feature modeling from FODA, such as featureRSEB and Cardinality-Based Feature Modeling (CBFM). While some other techniques focus on variability with use case, such as the research of Ma?en and Lichter, and Halmans and Pohl. Another variability technique such as Variability Specification Language (SVL), Koalish, Pure:Variants, CnnIPF, and COVAMOF introduced variability modeling used their own concepts.

Sinnema and Deelstra [MS07] worked in classifying variability modeling techniques based on a framework of key characteristic, such as how variability information is represented; how choices can be extended, changed, or customized; how the specific selection from the available options is represented; available tool support; etc. Based on those classification categories, Sinnema and Deelstra chose five variability modeling techniques to be compared; VSL, ConIPF, CBFM, Koalish, and Pure:Variants. Furthermore, the comparison shows the similarities and differences by exemplifying with a running example. The comparison is not meant to promote a specific technique, but rather than to focus on variability in requirements. The variability modeling classification identifies three important issues in variability modeling:

- Variability management is a complicated task, where many choices and constraints are involved.
- Most variability techniques lack a description of a process, which is re-

quired to have a successful deployment.

- Most techniques are based on a principle that requires a fully formalized variability model.

2.2.5 Variability Management in Software Product Line (SPL)

Implementation of variability concepts are mostly applied in software product line engineering, where the paradigm of variability in identifying commonalities of a product for reusability is really applicable. A Software Product Line is a set of products sharing a common architecture and a set of reusable components [SB00]. The architecture of software product line consists of a set of reusable components and a number of software products. A component in the SPL architecture implements a particular domain of functionality. A product is constructed by composing the components in the architecture. The approach used in software product line begins with selecting a set of products comprising a product line, and identify requirements that are common to all products (commonalities) and what differentiate them (variability).

Managing variability in software product line consists of the following tasks:

- Identifying the variability. The initial phase of the software product line development process is to analyze the requirements of a number of products. The aim of this process is to identify what is shared by all products and where the products differ.
- Introducing variability to the system. After variability is identified, the next process is to find variation points of a system and to choose a mechanism that will be used to implement the variability.
- Collecting the variants. The result of collecting the variants is a set of variants associated with a variation point.
- Binding the system to one variant. Binding the system is done by associating a particular variation point with one of its variants.

The aim of identifying variability in Software Product Line can be viewed in two dimensions; space and time. The space dimension is concerned with the use of common parts in multiple products, to minimize the cost of a product development. The time dimension is concerned with the ability of product to support evolution and changing requirements in various contexts, reduce the time to market of product distribution .

2.3 Enterprise JavaBeans (EJB) 3.0

2.3.1 EJB 3.0 Introduction

Enterprise JavaBeans (EJB) is a server side component architecture that encapsulates the business logic of an application. The EJB business applications are written in Java, scalable and can be deployed on any platform that supports the EJB specification. EJB was developed by IBM in 1997, and later adopted by Sun Microsystems in 1999. After few years enhancement under the Java Community Process EJB 3.0 was released in May 2006 with radical changes from previous version [Sik08]. One major change in EJB 3.0 is the handling of persistence that is no longer provided by an EJB container, but rather by a Java Persistence API (JPA) persistence provider. Other feature introduced in EJB 3.0 is metadata annotations.

EJB applications are deployed and run under the control of an EJB container within an application server. The EJB container provides common services needed in enterprise applications such as persistency, transactional integrity, security, and system management .

2.3.2 EJB 3.0 Artifacts

There are three main artifacts in business application of EJB technology: session beans, message-driven beans, and entities.

Session Beans

In EJB technology, encapsulation of business processes, such as book reservation, transfer funds are handled by session beans. These session beans are not persistent and not stored in a database or other permanent file system, but session beans can create and update entities which are persistent. There are two types of session beans: stateless and stateful. Stateless session beans are business objects that do not have a state associated with them, such as sending email to customer support as one-off operation and one step process. On the other hand, stateful session beans maintain a state for an object. The reference to a bean will end when the users end the session or the session times out. As mention before, session beans are transient, therefore the state is not written to a database but held in the containers cache. The common example for stateful session beans is an online shopping cart, where the customer's order will be maintained until the user finished or canceled the purchase, or until the session times out. Both types of session beans uses annotation to indicate the type of the beans (`@Stateless` or `@Stateful`).

Message-Driven Beans

Message-driven beans are business objects that are executed by messages instead of method calls. Message-driven beans are stateless, and like session beans, they can invoke other session beans and can interact with entities. An example for message-driven beans is a registration process: a session bean sends a message to a JMS queue requesting that a new customer be added to the database. The message will be a customer object with its detail information, the message-driven bean will simply add the customer to the database when receiving the message.

Java Persistence API (JPA)

In EJB 3.0, entities are persisted by a persistence provider or persistence engine implementing the JPA specification. JPA is a separate specification and application from the EJB container that provides some services for EJBs, such as the Entity Manager, object or relational mapping, the Java Persistence Language Query (JPLQ). The Entity Manager is provided for persistence, transaction management, and managing the lifecycle of entities. Object/Relational annotations provide the mapping for entities into relational database table and JPQL retrieves the persisted entities.

2.3.3 EJB architecture

EJB 3.0 is a part of Java EE version 5 (Java EE 5). Therefore, the EJB 3.0 architecture is based upon the Java EE 5 architecture with a 3-layer model: presentation layer, business layer, and database layer. The presentation layer presents the user interface and handles interactions with the end users. The business layer is responsible for executing the business logic. The data layer stores business data.

As shown in figure 2.5. [Sik08], EJB artifacts are placed in the business layer of the Java EE 5 architecture. The business layer was divided into two sub-layers, business logic layer and persistence layer. The business logic layer is concerned with business processing, where session beans and message-driven beans are deployed and executed in the EJB 3.0 container. The persistence layer handles entity artifacts, which are persisted to the database using the JPA persistence engine. The EJB architecture offers a standard for developing distributed, object oriented, component-based business applications. If EJBs (session and message-driven beans) have been well designed, they can be reused by other applications. EJBs are distributed in the sense that they can reside on different servers and can be invoked by a remote client form a different system on the network.

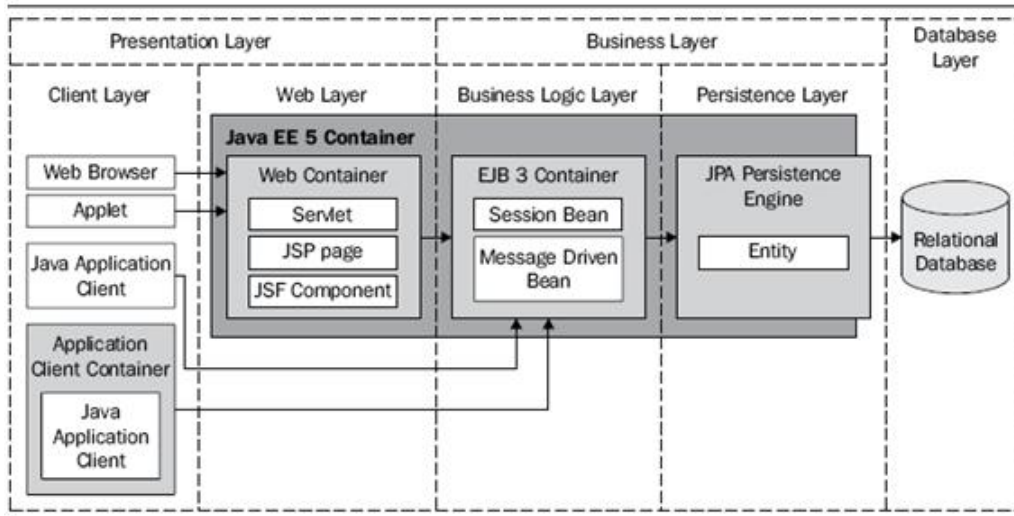


Figure 2.5: Java EE 5 Architecture

2.4 The MeDIC Information System

2.4.1 The MeDIC Introduction

MeDIC is a metric management software system to define, maintain, document, and develop metric. MeDIC was developed as a support tool to manage metric within a project or organization. A support tool to manage, communicate, and develop metrics will help to avoid errors, misinterpretations, or sporadic use of metrics, and help to share the knowledge from the metric experts.

Using Enterprise JavaBeans 3.0 as the base technology, the Research Group Software construction (SWC) of the RWTH Aachen developed a web-based software tool to support company's employees in the definition of new metrics and reuse of existing knowledge concerning metrics. This tool is used at Generali Deutschland Informatik Service GmbH who is a cooperation partner of the research group.

The specification process in this system is based on GQM and GAM methods as mention in section 2 .1.2. The first step is to specify the goals and the requirements of the measure. These requirements will be deducted, and later will be described as the information needs for metrics (as a question). The next steps are analysis, specification, modeling, and documentation to ensure that the metric will be able to answer the information need.

As mentioned in section 2.1.3, CMMI covers the use of metric in the Measurement and Analysis (MA) process area. The MeDIC system is designed to meet the following requirements of the CMMI:

- Establish and maintain measurement objectives that are derived from identified information needs and objectives.
- Specify measures to address the measurement objectives.
- Specify how measurement data will be Obtained and stored.
- Specify how measurement data will be analyzed and reported

2.4.2 Main processes in The MeDIC system

There are four main processes in the MeDIC system; managing project, standard metrics, project specific metrics, and information needs. Defining the project is the initial step of the MeDIC system, follow by proposing or choosing metrics. Metrics are divided into two type, standard metric and project specific metric. Standard metrics are the best practice metric used by the organization, each project can choose the available standard metrics that have same information need. The second types of metrics are the project specific metrics, which are created for each specific project purposes. Both of the metrics belong to at least one project in the system. Information needs are categorized in several types to manage metrics in organization. Project that needs metric with the same information need might reuse available metric.

Initiate and manage the project Create a new project by identify the project name, the user will automatically be assigned as Project Manager. Project Managers have the authorities to add other users into the project and manage the roles of all users, manage the project itself, including all metrics that belong to the project, and deactivate the project.

Manage the usage of standard metrics for a project Managing standard metrics starts by choosing one of the available standard metrics based on the information need, and adding this standard metric to the project. While adding the standard metric, some information needs to be redefined to adjust the general standard metric to the current project. All users will be able to see the adjustments and compare them to the default of the standard metric.

Manage the project specific metric Before a new project specific metric can be proposed, the information need has to be defined, by choosing the available information need or create a new one. The next step is then the definition of all the metric specification parameters.

Manage information needs Information needs are grouped in several categories. With this management, the user can choose the available standard metric easily based on their information need. Defining the information need before proposing a new metric will be easier as well.

2.4.3 Use case diagram

The use case diagram in figure 2.6 shows the overall use cases of the MeDIC system. There are four actors that are associated with the MeDIC system: guest, metric data provider, metric client, and metric coordinator. All actors have specific activity based on their privilege to manage projects, metrics, and information needs on the system as shown in the use case diagram below. The implementation of the MeDIC system is still in progress, not all cases have been implemented. The current implementation is the MeDIC version 2.0.

2.4.4 The MeDIC system architecture

The application architecture of the MeDIC system is based upon the Java EE 5 standard architecture (see section 2.4.1.). The architecture is divided into 3 parts: presentation layer, business layer and data layer as shown in figure 2.7. The presentation layer will provide the GUI for user to communicate with the system via browser. This presentation layer is divided into 2 parts, the JSPs (Java Server Pages) for HTML code and the servlets for the logical code.

The business layer is derived into 4 EJB parts: Application Facade, Controller, Management, and Entity. The separation between Entity and Management was elected to the entity management not to mingle with the persistence entities itself. The business logic is defined in the Controller, to control the Management and Entity bean. The application Facade forms abstractions from the Controller to ease the communicating with the presentation layer.

2.4.5 Implementation of the MeDIC

The system is designed using IBM Rational Application Developer for WebSphere Software version 7.5.3 (RAD), a commercial Eclipse-based integrated development environment (IDE). RAD V.7.5.3 provides Java Enterprise Edition (JEE) technology, including: Enterprise Java beans (EJB) applications for distributed, secure applications with transactional support, Java Persistence API (JPA) applications to access persistent data, and Java Server Pages (JSP) or Java Server Faces (JSF) for developing presentation logic. The application server, WebSphere Application Server (WAS) V.7.0 is the compatible server for the application environment that we used. And for the database, BD2 by IBM is used as our relational model database server.

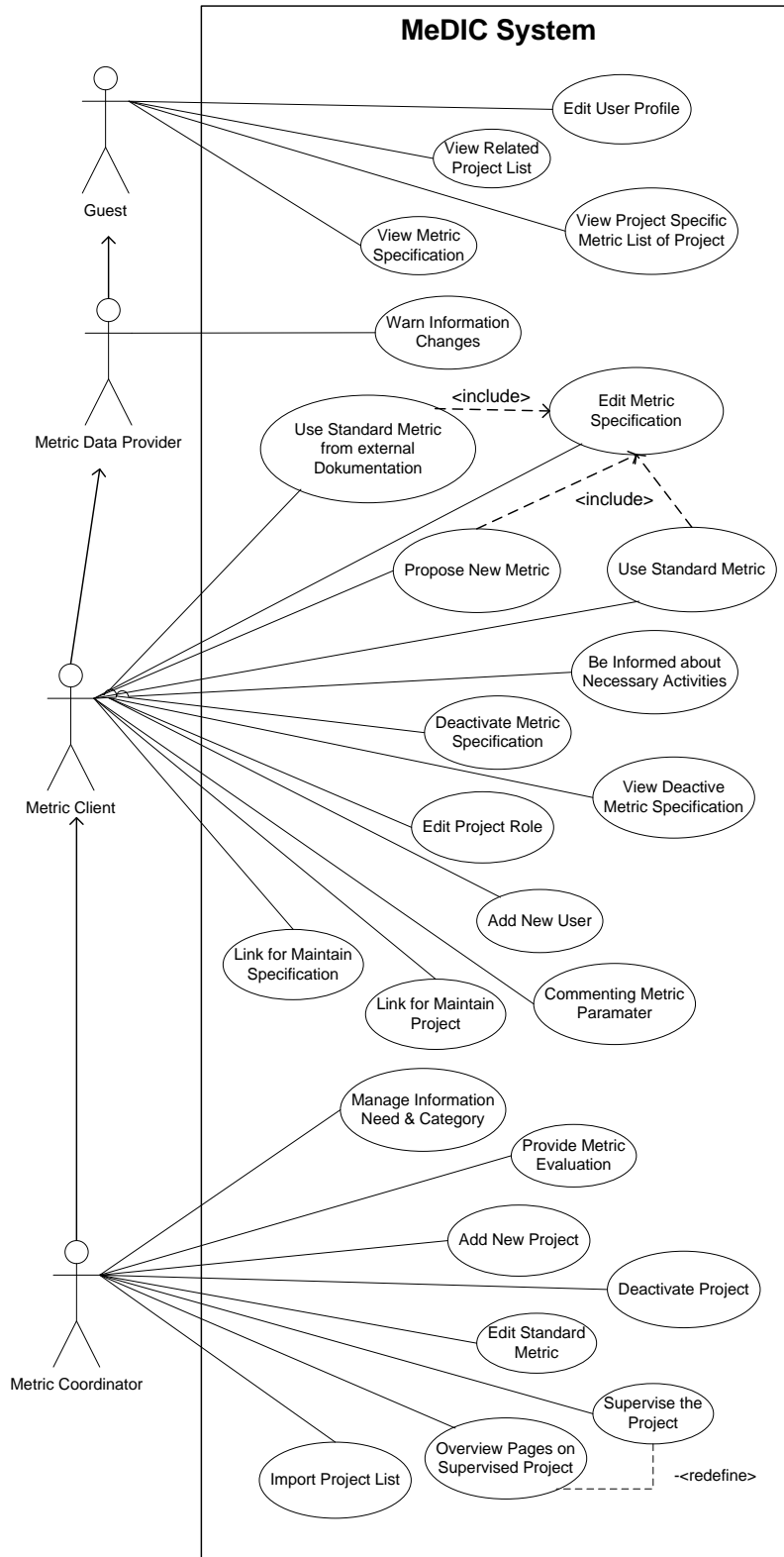


Figure 2.6: Use case diagram of the MeDIC system

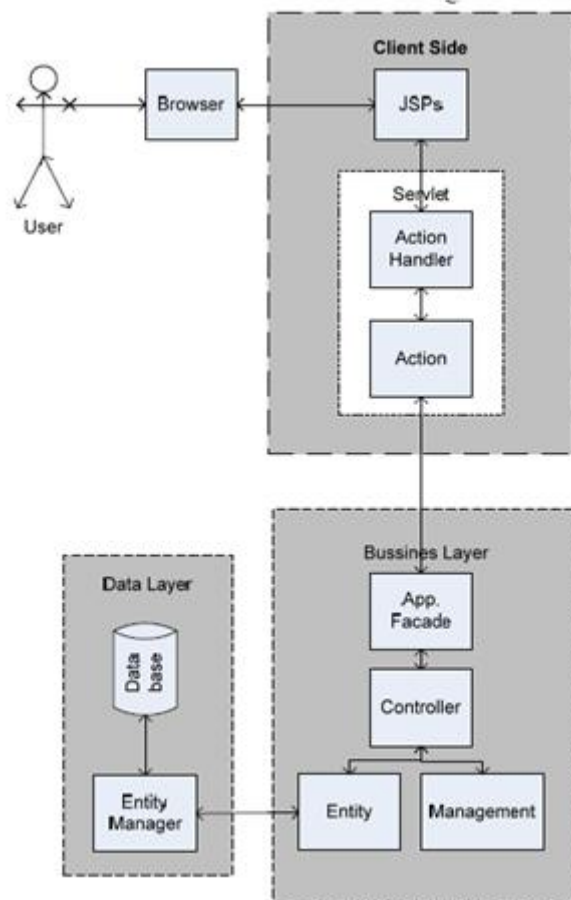


Figure 2.7: The MeDIC System Architecture

