

Designing for Increased Software Maintainability

Jun Han

Peninsula School of Computing and Information Technology
Monash University, McMahons Road, Frankston, Vic 3199, Australia
phone: +61 3 99044604, fax: +61 3 99044124, e-mail: jhan@monash.edu.au

Abstract

The recording and management of design rationales and design decisions are important issues in achieving better support for software maintenance. In the context of existing work on design rationale models, this paper introduces an approach to the integration of system design, design decisions and design rationales and the formalisation of design decisions in terms of software artifacts and their relationships. The relationships and properties characterising design decisions capture precisely the intent of these decisions, and can be checked and enforced to ensure that the decisions are followed and maintained in the actual development. We also show how the formalised design decisions can be used in better supporting software comprehension and maintenance.

1. Introduction

In the software design process, the design team usually considers a number of possible design choices and make a reasoned design decision for a given design issue. It has been widely recognised that recording the design decisions and design rationales can bring significant benefits to software comprehension and maintenance. Over the years, there have been a number of research efforts in developing methods and tools for recording and utilising design decisions and design rationales, including those reported in [13, 3, 2, 10, 1]. In general, these approaches set out, in a structured way, the design alternatives which were considered for a design issue, the arguments for and against these alternatives and the reasons that lead to the final design decisions [12]. As such, the recording and use of the design decisions and design rationales provide support for the comprehension and maintenance of the software system concerned.

While having achieved significant advance, the exist-

ing approaches to the recording and use of design decisions and design rationales can be further improved from the following two aspects:

1. Most of existing design rationale models and tools are developed and used separately from the design methods and tools. As such, the recording of design rationales and design decisions continue to be under-utilised in the industry [12]. To facilitate the full use of the recorded design rationales and decisions and realise their full benefits in software comprehension and maintenance, they must be integrated with the design specification so that the system design, design decisions and design rationales can be captured, manipulated, used and supported in a coherent manner. In particular, the final design decisions should be *linked* to the relevant parts of the design specification at appropriate levels of granularity.
2. Since the design rationales and design decisions are not appropriately integrated with the design specification, the properties characterising the design decisions are not formalised in terms of the design specification in the current approaches, so that they can not be used to check and ensure that the design decisions are followed in the actual design (specification). When formalised, the properties characterising design decisions can be used in the support tool to aid the software comprehension and maintenance process.

In this paper, we introduce an approach to formalising design decisions directly based on and integrated with, design representations (or specifications). As such, it is not meant to be an alternative to the existing approaches. Rather, it complements the existing approaches and can be integrated with them to achieve better support for software comprehension and maintenance from the above two aspects.

In our approach, the properties characterising design decisions are formalised as constraints on the rep-

resentation structures of the actual software design and implementation artifacts. As such, it provides better support for software comprehension in terms of *precise* characterisation of the design decisions, and for software maintenance in the sense that the constraints can be used for consistency checking, change impact analysis and change propagation.

In section 2, we introduce our approach to formalising design decisions. In section 3, we present a case study to illustrate the approach and the benefits it delivers for software comprehension and maintenance. After discussing related work in section 4, we conclude the paper in section 5.

2. Formalising design decisions

Our approach to the formalisation and use of design decisions is set in the context of a software engineering environment, so that the design decisions can be characterised based on the software artifacts (e.g., design and implementation documents) and their relationships in the environment. As such, not only the design decisions and rationales can be linked to the design specification (including all kinds of software artifacts), but also their (partial) meaning can be formalised to constrain the design specification and ensure that the design decisions are followed in the actual design and implementation. In this section, we first introduce the most relevant features of the underlying software engineering environment, and then present our approach to formalising design decisions.

2.1. Preliminaries

Artifact representation. The representation of software artifacts is the basis of our approach to software engineering environments¹. It has the following three major aspects [15, 4, 7, 5]:

- *Fine-grained representation of individual artifacts.* As in traditional language-processing tools such as compilers and language-based editors, we use syntax trees to represent software artifacts (or documents) and use an augmented EBNF notation to describe their syntactic structures (the EBNF productions are also called *segment type rules* as the artifact components are referred to as *artifact segments*) [4]. In addition, we allow representation and specification of multiple artifacts in the environment, ordered and un-ordered composition of

¹Other aspects of our approach such as artifact manipulation and presentation have less relevance to the current paper and are therefore omitted from this overview.

artifact segments, and artifacts with formal syntax or written in natural language.

- *Artifact relationships.* The relationships between software artifacts or their components are captured and represented in the environment as *structural relations* [4]. A structural relation connects one artifact (or artifact component) to another artifact (or artifact component). Each type of structural relations may connect certain types of artifacts (or artifact components). A given type of structural relations can be defined using *relation type rules* by giving the names of the relation type as well as the left and right hand side artifact (component) types connected. Also note that certain relation types may have associated attributes and content. Like artifacts, relationships have first-class status in our approach.
- *Artifact properties.* The segment and relation type rules explained above can be viewed as syntactic constraints on software artifacts and their relationships. Besides, additional *structural properties* or *constraints* may apply to software artifacts and their relationships as to their consistency [7], such as the scoping rules of different kinds of identifiers in a program. In general, these properties are specified in the form of Boolean expressions using certain conventions and primitive functions/predicates, and their enforcement and violation are managed by the environment [7].

To provide support for software development activities in the context of a given project, the environment allows the definition of the required artifact structures, relation types and artifact properties. Augmented with these specific requirements, the environment provides specialised support for the given project. In the next section, we will see examples of using these environment features.

Methodology-specific vs. application-specific requirements. The support requirements specific to a development project can be classified into two categories: those specific to the chosen development methodology, and those specific to the application being developed. The methodology-specific requirements concern the nature of the development methodology. In a development methodology using the Booch design method and the C++ implementation language, for example, we need to deal with structures of the various Booch design artifacts like class diagrams and state transition diagrams, structures of C++ programs, different types of relationships among the design and implementation artifacts, and the constraints

(or properties) of these artifacts and relationships. These methodology-specific requirements are independent of the application being developed. That is, the same requirements still apply if we use the methodology to develop a different application. As such, the methodology-specific requirements usually can be specified to specialise the generic environment before the actual development starts. So far we have applied our artifact representation techniques to a range of methodology-specific (but application-independent) aspects of project support [6, 8], i.e., defining the types of documents, the types of relationships and the type-based properties without reference to specific features of the application. These (static) definitions are used to augment the generic environment to formulate a methodology-specific environment which can be used in any project following that methodology [5].

The application-specific requirements depend on the nature of the actual application's development, and therefore can only be identified as the development progresses. Examples include the number of subsystems required, a specific relation between two system modules as dictated by the design, and an application-oriented property about a relation and the connected artifacts. To be able to provide support for these application-specific requirements, the environment needs to have the capability of being dynamically specialised with new (application-specific) artifact types, relation types and properties as the development continues.

Design decisions are application specific, and can only be formulated as the project progresses. That is, they can not be foreseen before the project actually starts. As such, the characterisation of design decisions in terms of software artifacts depends on the environment's capability of dynamic augmentation. The next subsection introduces our approach to the formalisation and use of (application-specific) design decisions based on this environment capability.

2.2. Formalisation and use of design decisions

Design decisions are made as a system's development progresses. These decisions are reflected in the design in a variety of ways. For example, one decision may concern the number of subsystems to be used, while another may deal with the selection of a particular inter-subsystem communication protocol. In this

²In general, dynamic augmentation of the environment may also involve modification and deletion of existing artifact and relation types as well as existing properties, which is an active research area. In this paper, however, we are only concerned with addition of such features as we concentrate on the formalisation and use of design decisions.

paper, we focus on those design decisions that are related to software artifacts. As such, we may formulate these decisions in terms of software artifacts and ensure that these decisions are followed (or enforced) in the initial development and subsequent maintenance of the software.

Supporting design decisions through dynamic environment augmentation. As discussed earlier, the application-specific design decisions can only be *dynamically* formulated and enforced by the environment as the development progresses. This requires that the developer using the support environment act in two capacities: one is the software engineer who develops the software system with the support of the development environment, and the other is the environment engineer who augments and specialises the development environment. For example, the developer may realise that there is a need to link one part to another part of the implementation document to convey a particular relationship. However, the development environment is not currently augmented to support this type of relationship. To satisfy the above requirement, the developer needs to (1) define a relation type in terms of the document component types to be connected to augment the environment – task of the environment engineer, and (2) introduce the actual relationship which is an instance of the newly defined relation type – task of the software engineer. Similarly, the developer may also introduce new document properties/constraints into the environment to reflect a design decision. Although less likely, the developer may introduce new types of software documents into the environment as required, including the type definitions and the actual documents themselves. In general, therefore, this utilises the environment capability of dynamic augmentation. In the following discussion, we concentrate on the use of this capability to support the formalisation and use of design decisions.

Formalising design decisions. Approaches to the support of design decisions and design rationales, such as those suggested in [13, 12], usually involve a model for the ordinary software artifacts capturing the development and a model for design rationales and decisions. These two models are related in the sense that

1. a *design issue* is raised relative to some software artifacts,
2. a design deliberation process is carried out according to the design rational model, and
3. a *design decision* is made which may affect a number of software artifacts.

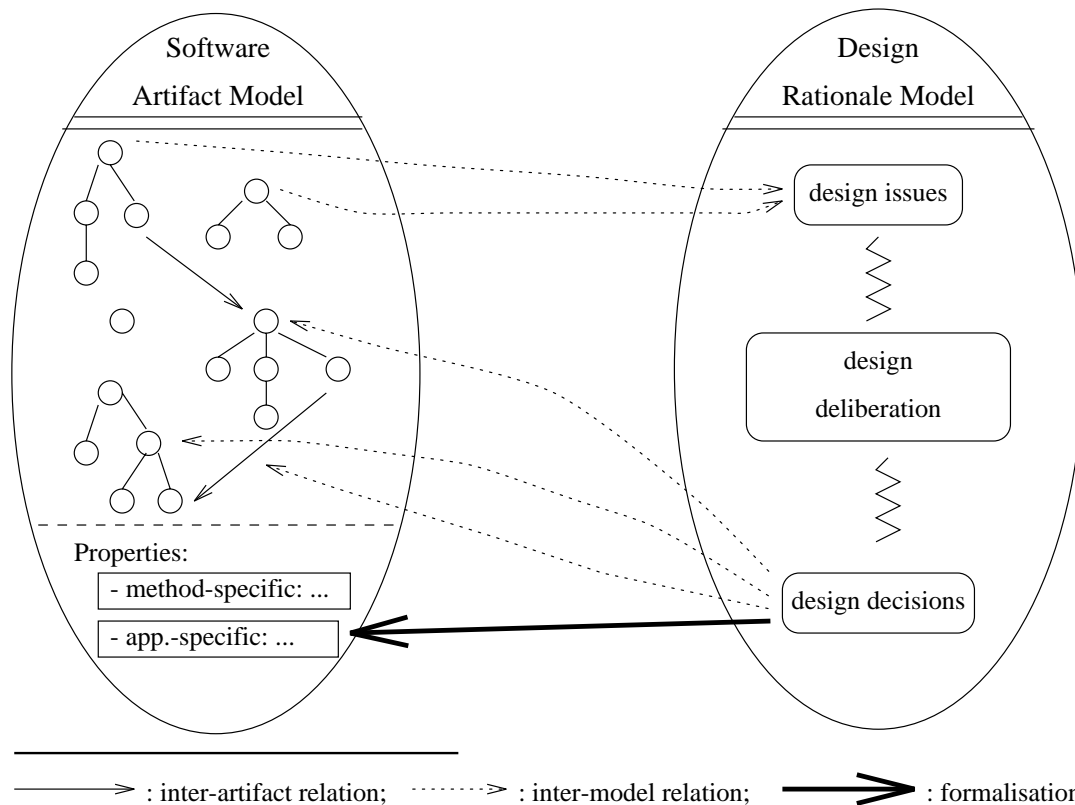


Figure 1. Relationships among software artifacts, design decisions and design rationales.

The existing approaches concentrate on the details of the design rationale model, and at best relate the design issues and decisions to the relevant software artifacts for traceability purposes (but at a coarse-grained level). The meaning and intent of a design decision are documented as part of the design rationale model, and are not formalised in terms of the software artifacts. Therefore, whether or not the design decision is accurately followed in the design can not be formally checked.

Our approach is based on the above general framework. Both the software artifact model and the design rationale model are represented in the environment. It adds to above framework the capabilities of (1) supporting inter-model relationships at an ydesired level of granularity, and(2) formalising design decisions in terms of software artifacts and their relationships (see Figure 1). The flexibility in defining relationships enhances the support for traceability between the models, and forms part of the basis for formalising and using design decisions. The formalisation capability allows us to formally state, check and enforce the characteristics of a design decision as properties of the software artifacts. While acknowledging that the meaning and intent of some design decisions can not be fully for-

malised, it is certain that those characteristics of the design decisions that are formalised can be used to ensure that the design decisions are followed and maintained during the system's development and maintenance.

A design decision is usually stated in relation to software artifacts, and their relationships and properties. Therefore, its formalisation will involve different document types, relation types and properties. Some of them may be already supported by the environment, but others may need to be newly defined. The latter concerns the dynamic augmentation of the environment (see above). Once the environment augmentation is done, the developer can introduce the actual documents and relationships, on which the properties characterising the design decision will apply.

Using design decisions. With the design decisions being formalised in terms of the software artifacts, the environment can provide better support for software comprehension and maintenance. For example, the software documents that are relevant to a design decision can be linked together using relationships, and the intent of the decision can be captured using properties.

In terms of software comprehension, the relationships and properties can facilitate browsing, navigation and reasoned traceability among the relevant artifacts (or artifact components). In particular, the properties capture precisely the requirements of the design decision. This enables that the developers always have a common understanding about the intent of the design decision. In terms of maintenance, the relationships and properties (constraints) can be used for consistency checking and enforcement to ensure that the design decision is followed. In particular, they can be used for change impact analysis and actual change propagation, so that the ripple effects of a change can be identified and the task of maintaining the entire system's consistency is made easier. In fact, all these benefits of formalised design decisions not only apply during the maintenance of a software system but also during its initial development. In the next section we will see examples of such use of formalised design decisions.

3. A case study

To demonstrate the formalisation and use of design decisions, we have carried out an extensive case study. The example software system (application) chosen in the case study is a weather monitoring system presented by Booch in [1]. Part of the methodology used to develop this application involves design in the Booch method [1] and implementation in C++. Due to space limitation and for simplicity, only a small relevant part of the case study is presented here.

The weather monitoring system is to provide automatic monitoring of various weather conditions, and in particular measuring wind speed and direction, temperature, barometric pressure and humidity. Further detailed requirements are not particularly relevant to this study and can be found in [1]. Figure 2 shows a partial design of the weather monitoring system, containing a class diagram and an illustrative state transition diagram [1]. The class diagram involves ten ordinary classes (excluding `RainfallSensor`) and three abstract classes, with association, has and inheritance relationships between them. The state transition diagram `IMstd` details the class `InputManager`'s state changes upon pressing different types of user input keys.

The implementation document is a C++ program, in which there is a C++ class for each design class in the class diagram above. Among these C++ classes, there are relationships mirroring the association, has and inheritance relationships between their corresponding design classes. In addition, other modules/segments may also be present in the implemen-

tation document, containing entities such as the main function and type declarations. In particular, the relationships between the declaration and use of a type can be explicitly captured.

Between the entities of the design document and the entities of the implementation document, there exist different types of relationships. In particular, there is a `designToImp` relationship between the entire design document and the entire implementation document, and a `hasImp` relationship between a design class and its corresponding implementation class. (Note that the implementation class also depends on the design class's state transition diagram, which is captured indirectly by the relevant `hasSTD` and `hasImp` relationships.)

Methodology-specific formulation. To capture some of the above methodology-specific characteristics, we have the following type definitions for the document structures and relationships:

```

DesignDoc = ClassDiag { STD } ...;
ClassDiag = { Class } { AbsClass } ...;

association =
    <Class | AbsClass, Class | AbsClass>;
has = <Class | AbsClass, Class>;
inheritance =
    <Class | AbsClass, Class | AbsClass>;
hasSTD = <Class, STD>;

ImpDoc = CppProgram;
CppProgram = { ImpClass } { ImpAbsClass }
    MainFunc { TypeDec } ...;
TypeDec = EnumDec | ...;
EnumDec = "enum" TypeName EnumValues;

impAssociation =
    <ImpClass | ImpAbsClass | MainFunc,
    ImpClass | ImpAbsClass>;
impInheritance = ...;
impHas = ...;
typeToUse = <TypeDec,
    ImpClass | ImpAbsClass | ...>;

designToImp = <DesignDoc, ImpDoc>;
hasImp = <Class, ImpClass> |
    <AbsClass, ImpAbsClass>;

```

Note that the above type definitions are only part of those required for the case study and are selected with the chosen application in mind.

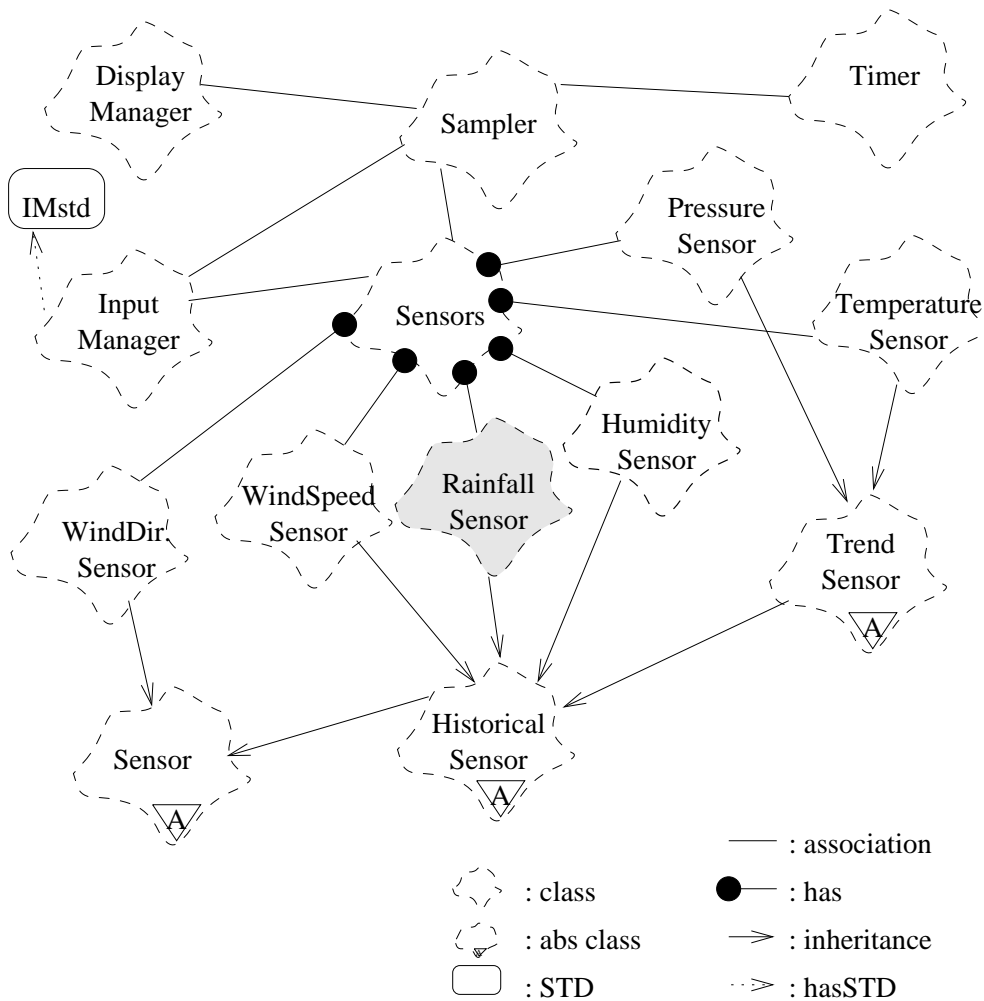


Figure 2. A partial design of a weather monitoring system.

Further characteristics of the development methodology can be formulated as properties of the artifacts and their relationships. For example, the requirement that a design class must have a corresponding implementation class can be formulated as a property `classMatch`: the mathematical relation formed from all the relationships of the type `hasImp` is total relative to all the design classes:

```
classMatch == total(hasImp, Class ∪ AbsClass);
```

The correspondence between the inter-class relationships at the design level and those at the implementation level can be formulated as a property `relMatch`:

```
relMatch == ((a, b) ∈ association ∧
  (a, c) ∈ hasImp ∧ (b, d) ∈ hasImp
  ⇒ (c, d) ∈ impAssociation)
  ∧ ((a, b) ∈ has ∧ (a, c) ∈ hasImp ∧
  (b, d) ∈ hasImp ⇒ (c, d) ∈ impHas)
```

```
∧ ((a, b) ∈ inheritance ∧
  (a, c) ∈ hasImp ∧ (b, d) ∈ hasImp
  ⇒ (c, d) ∈ impInheritance);
```

Augmented with the complete set of type rules and properties regarding the chosen development methodology (including those presented above), the generic environment becomes a software engineering environment specific to that methodology. With assistance of this methodology-specific environment, the software engineer may develop various applications, including the weather monitoring system.

An application-specific design decision. During the development of the weather monitoring system, various design decisions are made. One particular decision is to introduce a new type declaration enumerating the different kinds of sensors, so that it can be used by various implementation classes:

```
//Enumeration of sensor names
enum SensorName {Direction, Speed,
                 Temperature, Humidity, Pressure};
```

An example use of this type is that the implementation class `ImpDisplayManager` (corresponding to `DisplayManager`) uses it to discriminate the display functionality according to the sensor type. Therefore, there is a `typeToUse` relationship between the above type declaration and the implementation class `ImpDisplayManager`. Note that the methodology-specific formulation given above covers this consequence of the design decision, i.e., the developer does not need to introduce the definition of a new relation type. He or she can simply introduce the required relationship instance of the type `typeToUse`.

Following the above design decision, more importantly, we introduce a relationship of type `enumToImpClassH` between the above enumerated type declaration and the root of the inheritance hierarchy in the implementation document (i.e., `ImpSensor`) in order to indicate that the values in the enumerated type declaration reflect the ordinary classes on the inheritance hierarchy:

```
enumToImpClassH =
    <EnumDec, ImpClass | ImpAbsClass>;
```

Note that this new relation type is not due to the development methodology or the language used, but is rather due to the above application-specific design decision. After defining the above relation type, the developer may introduce the actual relationship instance to facilitate traceability between the type declaration and the inheritance hierarchy.

To further formalise or convey the intent of the design decision and make it more precise in terms of the relevant artifacts and relationships, we introduce the following property:

$$\text{enumProp} == (a, b) \in \text{enumToImpClassH} \Rightarrow \begin{aligned} &\#a.\text{enumValues} = \\ &\#\{c : \text{ImpClass} \bullet (c, b) \in \text{impInheritance}^+\} \end{aligned}$$

which states that if there is an `enumToImpClassH` relationship between an enumerated type declaration a and an implementation class b , then the number of enumeration values in a is equal to the number of ordinary implementation classes on the inheritance hierarchy rooted at b .

In general, the `enumToImpClassH` relation and the `enumProp` property *formally* capture the major implication of the above design decision (about introducing the enumerated type) in the design specification (including the design and implementation documents).

The introduction of the relationship and the enforcement of the property will ensure that the design decision is followed. For example, any violation to the `enumProp` property will result in inconsistency in the development, which may indicate that further changes to the development are required (see below for an example).

Use of the formalised design decision. In terms of software comprehension, we may follow the `enumToImpClassH` to navigate easily from the enumerated type declaration to the inheritance hierarchy vice versa in the process of understanding the system design and implementation, especially reflecting the above design decision. More importantly, the `enumProp` property captures precisely the intent of the design decision and enables the developers to gain an accurate appreciation of the design decision made earlier. This precise formulation is particularly important since the developers who later have the need to understand the design decision may not be those who made the decision.

As mentioned above, the property characterising the design decision can be used as a constraint on the development. During the initial development or subsequent maintenance, we may enforce this constraint to ensure that the decision is followed even in the event of changes. For example, an accidental or deliberate deletion of an enumeration value will result in the violation of the property `enumProp`. This will prompt the developer to inspect the enumerated type declaration and the inheritance hierarchy concerned and to restore consistency as appropriate.

Another particular use of the above formalised design decision is to facilitate change impact analysis and change propagation during software maintenance. As presented in [1], a possible change to the weather monitoring system is to measure an additional weather condition, rainfall. Among other modifications to the development is the one that introduces a new design class `RainfallSensor`, which inherits from the abstract class `HistoricalSensor` (see Figure 2). This initial modification in the design raises the need for a new corresponding implementation class `ImpRainfallSensor` according to the `classMatch` property and an inheritance relation between `ImpRainfallSensor` and `ImpHistoricalSensor` according to `relMatch`.

According to the design decision-related property `enumProp`, the introduction of `ImpRainfallSensor` into the inheritance hierarchy in the implementation document further raises the need to modify the enumerated type declaration, i.e., to add a new enumeration value denoting the new sensor. The need

for this consequent modification can be automatically identified based on the `enumProp` property and the `enumToImpClassH` relation during *change impact analysis*. The relevant change pattern for this impact relationship is as follows:

$$\begin{aligned} &\{(a : \text{ImpClass}, \text{introduction})\} \rightarrow \\ &\{(b : \text{EnumDec}, \text{update})\} : \\ &(b, c) \in \text{enumToImpClassH} \wedge \\ &((a = c) \vee (a, c) \in \text{impInheritance}^+); \end{aligned}$$

which states that the *introduction* of an implementation class a will lead to the *update* of an enumerated type declaration b if a is on the inheritance hierarchy that is related to b via an `enumToImpClassH` relation. During *change propagation* the violation of the property `enumProp` after the earlier changes will guide the developer to make the consequent change on the enumerated type declaration, so that the property can be re-established and the initial design decision is (again) being enforced. Details of our strategy for software change impact analysis and change propagation are outside the scope of this paper, and can be found in [8].

In summary, this case study has shown that the formalisation of design decisions in terms of software artifacts and their relationships can be used to facilitate browsing, navigation, reasoned traceability, precise understanding, consistency checking and enforcement, change impact analysis and change propagation. These capabilities are applicable in both initial development and consequent maintenance of a software application, due to the fact that the development and maintenance are carried out in the same support environment.

4. Related work and discussions

Early work on design rationale modelling, such as gIBIS [3] and DRL/SYBIL [11, 9], concentrates on the argumentation process and model during system design. The models involve a range of concepts such as goals, issues, positions and arguments as well as their relationships. DRL/SYBIL also uses decision metrics in weighing the alternative positions. However, these early approaches are not particularly concerned with the relationships between design argumentation and actual design artifacts, which are important in the understanding of the development.

More recent systems dealing with design rationales and design decisions not only include a design rationale model, but also support the linking between the design rationale model and the software artifact model repre-

senting the actual design. These systems include Design Recording [13], REMAP [14], ED-LOD [2], DRCS [10] and Proteus [12]. The rationale models in these systems are more sophisticated than earlier systems with richer semantics towards the support for software development. Some of them are specifically tailored for a particular domain or aspect of (software) system development, e.g., DRCS for concurrent engineering and REMAP for requirements engineering. To a varied degree, they facilitate the traceability between the actual development/design (captured by a software artifact model according to a development methodology) and the decision-making process (represented through a design rationale model). In particular, Proteus aims to provide a practical and easy-to-use system for design rationale support by adapting existing models.

Our work reported in this paper builds on the existing work on design rationales and design decisions, and provides *further* support for software comprehension and maintenance by formalising design decisions in terms of software artifacts and their relationships. It is set in the context of a software engineering environment, where the software artifact model, the design rationale model and the inter-model relationships can all be represented. Since the environment supports fine-grained representation of documents and their relationships, the inter-model relationships can be formulated at any desired level of granularity. The environment's capability of dynamic augmentation forms the basis for the formalisation of application-specific design decisions as the development progresses.

5. Conclusions

In this paper, we have introduced an approach to formalising design decisions in terms of software artifacts and their relationships. Our approach is set in the context of existing work on design rationales and design decisions, and focuses on supporting the traceability (or relationships) among the software design, design rationales and design decisions and on formalising design decisions as properties of the development. Its automated support is based on the dynamic augmentation capability of a software engineering environment.

With formalised design decisions, we are able to achieve better support for software comprehension and maintenance. In terms of software comprehension, the relationships and properties that characterise design decisions facilitate browsing, navigation and traceability among the relevant artifacts. The properties capture precisely the intent of design decisions and aid their comprehension. In terms of maintenance, the relationships and properties can be used for consistency

checking and enforcement, to ensure that the design decisions are followed during initial development and subsequent maintenance. In particular, they can be used for change impact analysis and change propagation, so that the ripple effects of a change can be identified and the task of maintaining the entire system's consistency is made easier.

The case study reported in this paper has been completed. We are currently examining other typical case studies to refine our approach. At the same time, work on direct tool support for the approach is being carried out in our software engineering environment. Other issues for further investigation include support for changes to design decisions and closer integration with support for software change management.

Acknowledgment. I would like to thank my colleagues at Monash University and the anonymous reviewers for their valuable comments. The work reported here was partially funded by the Australian Research Council.

References

- [1] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.
- [2] A. Cimitile, F. Lanubile, and G. Visaggio. Traceability based on design decisions. In *Proceedings of 8th Conference on Software Maintenance*, pages 309–317, Orlando, USA, November 1992.
- [3] J. Conklin and M. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331, 1998.
- [4] J. Han. Software documents, their relationships and properties. In *Proceedings of 1st Asia-Pacific Software Engineering Conference (APSEC'94)*, pages 102–111, Tokyo, Japan, December 1994. IEEE Computer Society Press.
- [5] J. Han. A document-based approach to software engineering environments. In *Proceedings of 5th International CASE Symposium*, pages 128–133, Changsha, China, October–November 1995.
- [6] J. Han. Providing configuration management support in software engineering environments. In *Proceedings of 2nd Asia-Pacific Software Engineering Conference (APSEC '95)*, pages 282–291, Brisbane, Australia, December 1995. IEEE Computer Society Press.
- [7] J. Han. Specifying the structural properties of software documents. *Journal of Computing and Information*, pages 1333–1351, 1995.
- [8] J. Han. Supporting impact analysis and change propagation in software engineering environments. In *Proceedings of 8th International Workshop on Software Technology and Engineering Practice (STEP'97/CASE'97)*, to appear, London, UK, July 1997. IEEE Computer Society Press.
- [9] P. Jarczyk, P. Löffler, and F. Shipman. Design rationale for software engineering: A survey. In *Proceedings of 25th International Conference on System Sciences*, pages 577–586, Hawaii, 1992.
- [10] M. Klein. Capturing design rationale in concurrent engineering teams. *IEEE Computer*, pages 39–47, January 1993.
- [11] J. Lee and K. Lai. What's in design rationale? *Human-Computer Interaction*, 6(3-4):251–280, 1991.
- [12] S. Monk, I. Sommerville, J. Pendaries, and B. Durin. Supporting design rationale for system evolution. In *Proceedings of 5th European Software Engineering Conference (LNCS-989)*, pages 307–323, Sitges, Spain, September 1995.
- [13] C. Potts and G. Bruns. Recording the reasons for design decisions. In *Proceedings of 10th International Conference on Software Engineering*, pages 418–427, Singapore, April 1988. IEEE CS Press.
- [14] B. Ramesh. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, 1992.
- [15] J. Welsh and J. Han. Software documents: Concepts and tools. *Software — Concepts and Tools*, 15(1):12–25, 1994.