

ABSTRACT

Title of Document: METHODODOLOGY AND SOFTWARE
 PROTOTYPE FOR ONTOLOGY-ENABLED
 TRACEABILITY MECHANISMS

Cari Bever, Master of Science in Systems
Engineering, 2006

Directed By: Dr. Mark Austin, Associate Professor, Civil &
 Environmental Engineering; Affiliate Associate
 Professor, Institute for Systems Research;
 University of Maryland at College Park

Due to the rapid advancement of technology, industrial-aged systems are being replaced by information-based models through system integration, where hardware and software are combined by a variety of communication means. As engineering systems become progressively complex, the challenge is to fully understand and implement the connectivity relationships among various models of visualization so that catastrophic and expensive failures of engineering systems can be avoided. In order to achieve these connectivity relationships, this project inserts a new notion called “Design Concepts” in the traceability link between the already connected requirements and engineering objects, where rule-checking may be embedded into the design concepts. A software prototype of the Washington, D.C. Metro System has been built to illustrate the feasibility of connectivity between requirements, UML class diagrams and an engineering model. The software makes use of listener-driven events, which are a scalable and efficient method for establishing traceability links and responding to external user events.

METHODOLOGY AND SOFTWARE PROTOTYPE FOR ONTOLOGY-ENABLED
TRACEABILITY MECHANISMS

By

Cari E. Bever

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science in Systems Engineering
2006

Advisory Committee:
Dr. Mark Austin, Chair
Dr. Jeffrey Herrmann
Dr. Guangming Zhang

© Copyright by
Cari E. Bever
2006

Acknowledgements

This project would not have been possible without the support of many people. Many thanks to my advisor, Dr. Mark Austin, for his continuous guidance and encouragement throughout my entire thesis project. Also, thanks to Dr. Jeffrey Herrmann and Dr. Guangming Zhang for serving on my thesis committee. Thanks to my mother, Shirley Bever, for all the time she spent proofreading my paper for grammatical errors and also for the never-ending support and encouragement she provided. I would also like to thank Natasha Shmunis and Nicholas Oben for their help with the programming. Finally, I would like to thank my employer Raytheon for giving me the opportunity to receive my Master's degree in Systems Engineering.

Table of Contents

Chapter 1 Introduction	1
1.1 Problem Statement	1
1.2 Traceability Mechanisms and Models	5
1.3 State-of-the-Art Requirements Modeling and Visualization.....	7
1.4 Proposed Approach	10
1.5 Smart Building Application.....	12
1.6 Ontologies and Ontology-Enabled Computing.....	21
1.7 Framework for Interactive Ontology-Enabled Design.....	25
1.8 Project Scope and Objectives	28
Chapter 2 Design Concept Modeling with UML.....	30
2.1 Development of System-Level Design Representations.....	30
2.2 Representing Ontologies with UML Class Diagrams	34
2.3 Relationships between Classes.....	37
2.4 Human-Ontology Interaction and Processing	42
Chapter 3 Software Architecture Design.....	44
3.1 Software Architecture Design	44
3.2 Graphical User Interface (GUI) Design and Traceability Support	45
3.3 Listener-Driven Events Using the Java DEM.....	48
3.4 Violet UML Editor.....	55
Chapter 4 Application: Washington, D.C. Metro System.....	57
4.1 Washington, D.C. Metro System.....	57

4.2 Requirements Panel.....	58
4.3 UML Class Diagram Panel.....	59
4.4 Metro System Engineering Model Panel.....	61
4.5 Traceability Connections.....	64
4.5.1 Interaction with the Requirements Panel.....	65
4.5.2 Interaction with the UML Panel.....	67
4.5.3 Interaction with the Engineering Drawing Panel	68
Chapter 5 Conclusion.....	70
5.1 Conclusions	70
5.2 Future Work.....	71
Bibliography	76

List of Figures

Figure 1.1: Meta-model for Low-End Users of Traceability.....	6
Figure 1.2: Modeling Translational Mappings Across Hierarchies in SLATE	8
Figure 1.3: Modeling and Graphical Representation of Translational Mappings (TRAMs) in SLATE.....	9
Figure 1.4: Inheritance and Blocking Mechanisms in SLATE.....	10
Figure 1.5: Simplified View of State-of-the-Art Traceability and Proposed Model ..	11
Figure 1.6: Modeling of a Wireless Network in a Smart Building.....	13
Figure 1.7: Progressive Decomposition of Architectural Floorplans.....	15
Figure 1.8: Framework for Multi-Level Development for Building Architectures Augmented with Network Services.....	16
Figure 1.9: Pathways from Requirements to UML Representations of System Concepts to Multi-Level Representations of Building Architecture.....	19
Figure 1.10: Semantic Web Layer Cake Technologies and their Maturity	24
Figure 1.11: Present-day and Proposed Graphical User Interfaces.....	25
Figure 1.12: Multiple Conceptual Interpretations of a Single Design Object	27
Figure 1.13: Schematic of Viewpoint Interaction.....	28
Figure 2.1: Magic Square of System Development	31
Figure 2.2: Definition of Classes for Software and System-Level Design	32
Figure 2.3: Hierarchy and Network Abstractions for SE Development.....	32
Figure 2.4: Hierarchy and Network Abstractions for SE Development.....	33
Figure 2.5: Pathway from Meta-meta-models to Engineering Models and Systems..	35
Figure 2.6: Roles of the Meta-Meta-Model and Meta-Model in System Assembly...	36
Figure 2.7: Meta-model for Association and Link Relationships.....	37
Figure 2.8: Binary Associations in UML: Meta-Model and Example.....	38
Figure 2.9: Association Relationships in UML	39
Figure 2.10: Multiplicity Relationships in UML	40
Figure 2.11: Inheritance, Composition and Aggregation Relationships in UML.....	41

Figure 2.12: Connectivity between the Specification and Physical Models	43
Figure 2.13: Step-by-Step Implementation of Ontology Processing Machine	43
Figure 3.1: Overview of System Architecture Requirements.....	45
Figure 3.2: GUI Layout	46
Figure 3.3: Connection of XML to the GUI.....	46
Figure 3.4: One Listener Registered to One Source.....	49
Figure 3.5: Many Listeners Registered to one Source	50
Figure 3.6: One Listener Registered to Many Sources.....	50
Figure 3.7: Circles Example utilizing Event Listeners.....	52
Figure 3.8: System Response for Mouse Over "Small Circle" Rectangle.....	54
Figure 3.9: System Response for Mouse Over Small Circle	55
Figure 3.10: Example of a Class Diagram from Violet.....	56
Figure 4.1: Washington, D.C. Metro System (Washington, 2006).....	57
Figure 4.2: Washington, D.C. Metro System Application - Requirements Panel, UML Class Diagram, and Engineering Drawing.....	58
Figure 4.3: Requirements in XML	59
Figure 4.4: UML Class Diagram of Metro System.....	60
Figure 4.5: UML Class Diagram Saved in XML.....	61
Figure 4.6: Engineering Drawing of the Washington, D.C. Metro System.....	62
Figure 4.7: XML Fragment for D.C. Metro Stations	63
Figure 4.8: XML Fragment for D.C. Metro Tracks	63
Figure 4.9: XML Fragment for D.C. Metro Lines	64
Figure 4.10: Tracing Requirements to UML Diagram and Engineering Model.....	66
Figure 4.11: Verifying the First Requirement.....	67
Figure 4.12: Tracing UML Class Node to Objects in Engineering Model and Requirements.....	68
Figure 4.13: Tracing Engineering Model to UML Diagram and Requirements.....	69
Figure 5.1: Screenshot of the PaladinRM Graphical User Interface (GUI).....	74
Figure 5.2: Definition and Evaluation of System Assembled from Simple Shapes ...	74
Figure 5.3: Definition and Evaluation of a Simple Assembly	75
Figure 5.4: Definition and Evaluation of Spatial Relations Among Shapes	75

List of Tables

Table 1.1: Types of Diagrams in UML2 and SysML.....	4
Table 3.1: Examples of Actions Performed and Their Associated Event Listeners ...	49
Table 4.1: Table of Requirements.....	59

Chapter 1 Introduction

1.1 Problem Statement

The post-event analysis of recent engineering system failures indicates that, often, the underlying cause of catastrophic and expensive failures is minor mistakes or omission in communication of design intent (e.g., errors in the use of engineering units; errors in the placement of electronic devices on a drawing; logic errors in the implementation of software). The importance of this problem stems from the wide array of engineering applications that have failed in this way. Examples include spacecraft, automated baggage handling systems at airports, and networked services in modern building environments (Jackson, 2006; Jones, 2004; Sawyer, 1999). The difficulty in finding a good solution to this problem is due, in part, to industrial-age systems being replaced by information-age systems. As pointed out by Whitney (Whitney, 1996) in the mid 1990s, industrial-age systems tend to be dominated by hardware. Most hardware systems have continuous behavior that can be described by differential equations; generally, a small change in an input parameter will lead to a finite change in the system response. Designers can use safety factors to deal with uncertainties in system properties, behavior and design. Information-age systems, in contrast, tend to be dominated by combinations of hardware, software and communications, which together are required to provide new types of services, superior levels of performance, and work correctly with no errors. There are a number of reasons why satisfaction of these goals can be particularly difficult. First, when new

technologies are weaved together to achieve new types of functionality, systems can fail in new and unprecedented ways. In the late 1990s, for example, NASA certainly did not anticipate that a miscommunication of engineering units would lead to one of their spacecraft crashing into the surface of Mars. Second, correct functionality for software and communications systems is defined by logic (not differential equations). Not only does the concept of “safety factors” not apply, but as observed in a number of engineering system failures, a small fault in the software implementation can trigger (or result in) system level failures that are very costly and, sometimes, even catastrophic. At a first glance it is tempting to think “well, most errors are caused by bugs in the software – they can be fixed with more careful programming.” Jackson (Jackson, 2006) reports, however, that almost all grave software problems can be traced back to conceptual mistakes made before the programming was initiated.

This project is motivated by the premise that solutions to this class of problems will require better techniques for the representation, communication, visualization, and evaluation of engineering requirements attached to multidisciplinary engineering models (mathematical abstractions) and drawings (visual abstractions). These new techniques will need to operate within the bounds of established engineering and systems engineering practices – multi-disciplinary team-based design; use of multiple representations; support for multiple viewpoints (Peak, 1998). The CTO of Bentley Systems, a leading provider of architectural design software, notes that architectural/engineering firms need to move from “drawings” to “building information models,” the latter being capable

of representing and reasoning with graphical and non-graphical entities. Building information models are compelling because they enable processes for designing-in-context across disciplines and automatically enforcing standards. The resulting product is more correct by design (Bentley, 2003).

Assessment of Systems Engineering Community Response. In an effort to improve the accuracy and effectiveness of communication among engineers, the systems engineering community is working toward the development of SysML, the Unified Modeling Language (UML) extended for Systems Engineers (SysML, 2003; Unified, 2003). UML has already found great success in the software engineering community. By introducing a variety of new diagram types to SysML (e.g., requirements diagram; process diagram, as shown in the lower half of Table 1.1), the hope is that similar success will occur in systems engineering. Certainly the exchange of information between systems/requirements engineering tools can be improved through SysML and standards like AP 233 (Muller, 2003; Oliver, 2002).

In the long term, however, the name “Unified Modeling Language” promises more than it can (or should or will) ever deliver. While modest extensions to UML will be useful for documentation, informal analysis, and communication of ideas among systems engineers, UML lacks the syntax/semantics needed for rigorous analysis and formal verification of system compliance associated with temporal and spatial analysis of physical systems. While diagrams may represent different views on a system, there is no mechanism to define the interconnections or dependencies among the diagrams describing a system. In other words, there

are too many places to capture information (in the large number of available diagrams), and too few ways to show relationships between the diagrams (Berkenkotter, 2003). Moreover, recent history tells us that the benefits of UML are unlikely to be appreciated by upper-level management and discipline-specific engineers – instead, issues need to be explained in terms with which they are already familiar.

Part 1. Diagrams in UML 2

Structure Diagrams	Behavior Diagrams
Class Diagram	Activity Diagram
Component Diagram	Use Case Diagram
Object Diagram	State Machine Diagram
Composite Structure Diagram	Interaction Diagrams
Package Diagram	Sequence Diagram
Deployment Diagram	Communications Diagram
	Timing Diagram
	Interaction Overview Diagram

Part 2. Diagrams in SysML

Structure Diagrams	Behavior Diagrams
Block Diagram	Activity Diagram
Block Definition Diagram (extends UML Class Diagram)	(extends UML Activity Diagram)
Internal Block Diagram (extends UML Composite Structure Diagram)	Use Case Diagram
Parametric Constraint Diagram	State Machine Diagram
Parametric Definition Diagram	Sequence Diagram
Parametric Use Diagram	
	Cross-Cutting Diagrams
	Allocation Diagram
	Package Diagram (extends UML Package Diagram)
	Requirement Diagram

Table 1.1: Types of Diagrams in UML2 and SysML. (1) Structure and Behavior Diagrams in UML 2; (2) Structure, Behavior, and Cross-Cutting Diagrams in SysML

These gaps will not be bridged unless a method is found to use UML (and its extensions) in concert with discipline-specific models and notations (e.g., visualization of requirements; block diagrams; two- and three-dimensional engineering schematics). Therefore, the key tenet of the proposed work is that *end-to-end development of engineering systems will occur through multiple models of visualization networked together*. Looking ahead, there will still be a need for development of web-centric, graphically driven, computational platforms dedicated to system-level planning, analysis, design and verification of complex multidisciplinary engineering systems. These environments will employ semantic descriptions of application domains and use ontologies to enable validation of problem domains and communication (or mappings) among multiple disciplines. The abstraction of multiple disciplines to properly annotated information representations and reuse of previous work at all levels of development will be essential. Present-day systems engineering methodologies and tools are not designed to handle projects in this way.

1.2 Traceability Mechanisms and Models

As engineering designs become progressively complex, the task of understanding the connectivity relations among various parts of the design becomes increasingly difficult. Traceability mechanisms allow for an understanding of how and why various parts of the system development process are connected, thereby providing the development team greater confidence in: (1) Meeting objectives; (2) Assessing the impact of change; (3) Tracking progress;

and (4) Conducting trade-off analysis of cost against other measures of effectiveness.

Low-End Traceability Models. Industry experience indicates that low-end users of traceability tend to have problems that require less than 1,000 requirements (viewed as a mandate from the project sponsors or for compliance with standards) (Balasubramaniam, 2001).

As illustrated in Figure 1.1, users view traceability as a transformation of requirements documents to design. Before this can happen, however, requirements must be derived from higher-level system requirements. In the compliance verification phase of systems development, low-end users employ the requirements database to develop System Compliance Verification Procedures. Typically, low-end users lack support for grasping rationale for requirements issues and how they are resolved.

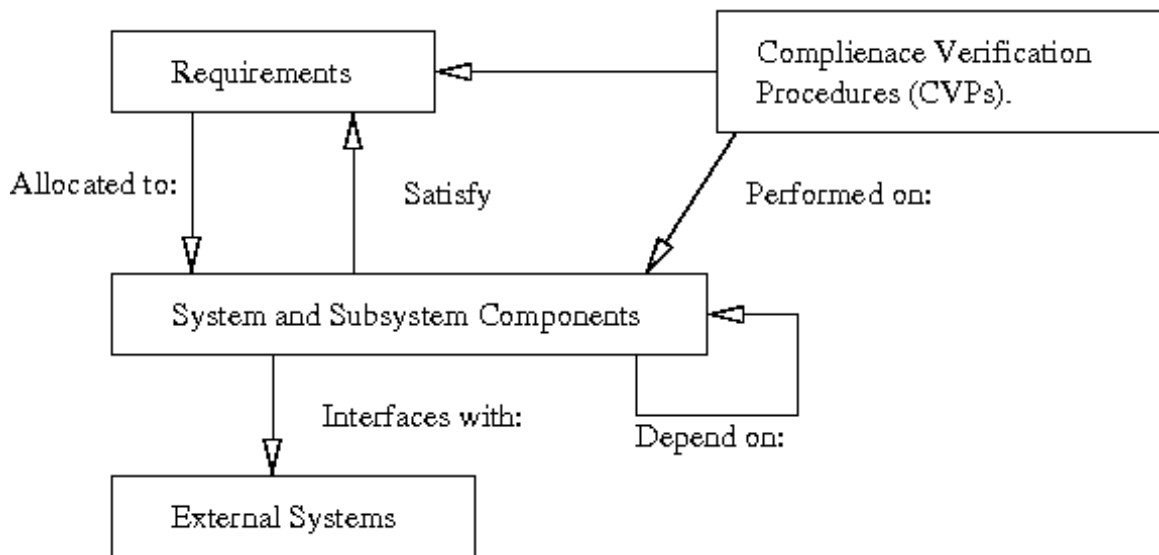


Figure 1.1: Meta-model for Low-End Users of Traceability

High-End Traceability Models. Industry experience indicates that high-end users of traceability tend to have problems that require, on average, about 10,000 requirements (viewed as a major opportunity for customer satisfaction and knowledge creation throughout the system lifecycle) (Balasubramaniam, 2001). They view traceability as an opportunity to increase the probability of producing a system that meets all customer requirements, is easier to maintain, and can be produced within cost and on schedule. High-end traceability employs much richer schemes of traceability (e.g., capture of discussion issues, decisions and rationale product-related and process-related dimensions) than their low-end counterparts. Traceability pathways of rationale enable accountability (e.g., what changes have been made; why and how they were made), particularly to stakeholders not directly involved in creation of the requirement.

1.3 State-of-the-Art Requirements Modeling and Visualization

In an ideal setting, project participants should be able to view design data/information in a manner with which they are familiar and easily understand “connectivity relations” and transitions among viewpoints and the rationale for establishing the connections in the first place. Unfortunately, state-of-the-art capability in requirements modeling and visualization falls well short of this vision.

As a case in point, the Teamcenter (SLATE) Requirements Tool is based upon very good data representations for connecting cause-and-effect relationships among “abstraction blocks” (ABs) in multiple viewpoints (i.e., TRAMS ==

translational mappings). Figure 1.2 shows, for example, the use of TRAMs to link requirements, electrical, mechanical and software viewpoints. The underlying graphical support is weak in the sense that all design entities are simply referred to as abstraction blocks (ABs). Moreover, to date, no one has been able to figure out how to actually organize and visualize the subsystem viewpoints on a computer as illustrated in Figure 1.2.

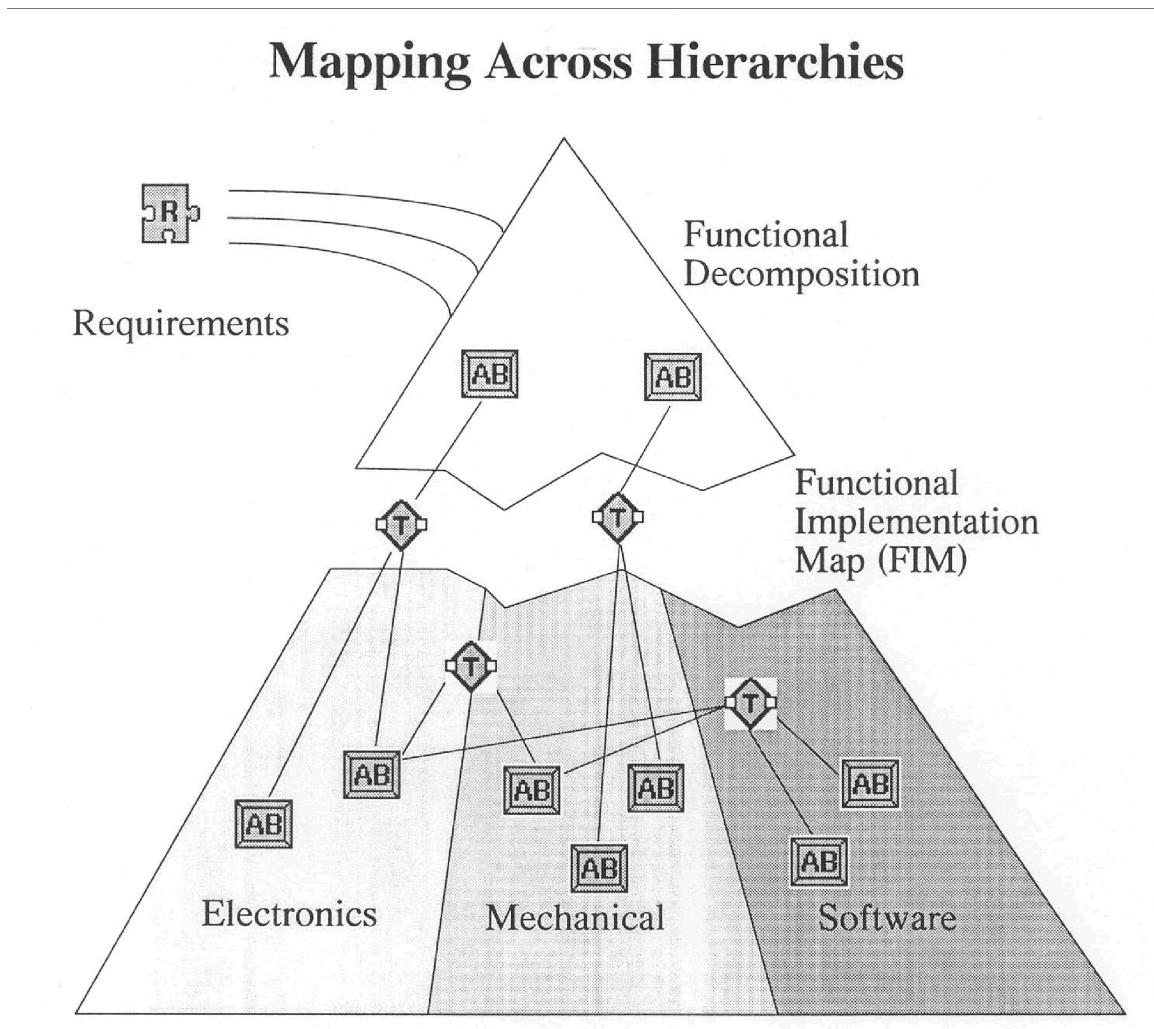


Figure 1.2: Modeling Translational Mappings Across Hierarchies in SLATE. The example shows use of TRAMs to link Requirements, Electrical, Mechanical and Software Viewpoints

Figures 1.3 and 1.4 show how TRAMs work in terms of connecting source ABs to destination ABs, and source-to-destination and destination-to-source pathways. Because the underlying graphical formalism is weak, many questions that a user might want to ask about requirements and/or the system structure remain unanswered or omitted. Simple questions like “Show me all complying and defining requirements that are related to this particular requirement” cannot be answered. Furthermore, a non-systems engineer is provided with no visual assistance in understanding how elements of his or her domain link to other domains.

To overcome these limitations there is a need for a better representation of individual objects (requirements, abstract blocks, etc.) and the linkage of those in the overall architecture design.

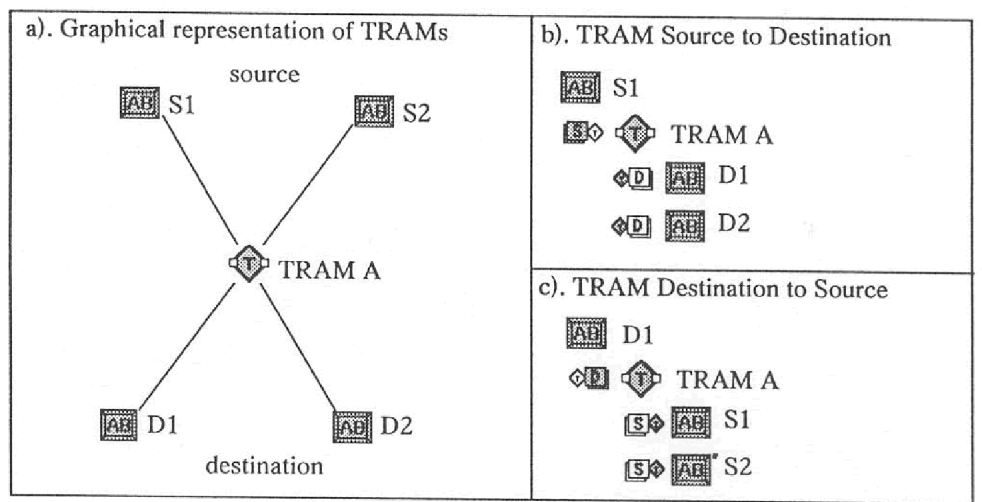


Figure 1.3: Modeling and Graphical Representation of Translational Mappings (TRAMs) in SLATE

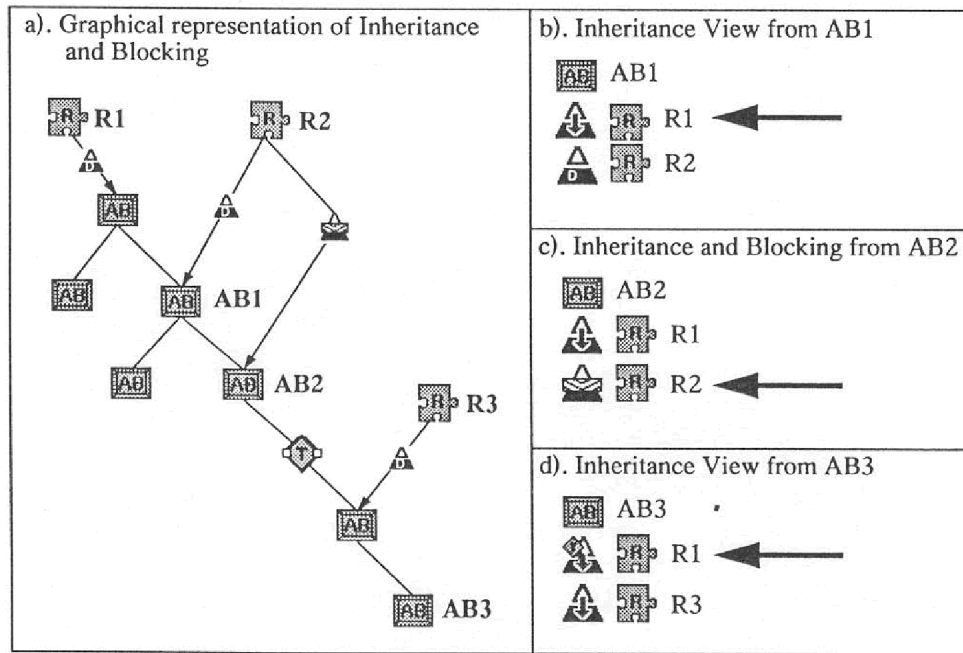


Figure 1.4: Inheritance and Blocking Mechanisms in SLATE. Three cases: (1) Abstraction block 1 complies with requirements R1 and R2, (2) Abstraction block 2 complies with requirement R1, but not R2, (3) Abstraction block 3 is defined by R3 – it also complies with R1 through the TRAM mechanism.

1.4 Proposed Approach

The upper half of Figure 1.5 shows a simplified representation for how requirements are connected to design elements in state-of-the-art traceability (i.e., traceability links connect requirements directly to design objects). State-of-the-art traceability mechanisms portray that a specific requirement is satisfied by a specific design object (or group of design objects). Or alternatively, looking backwards, this specific design object is included because it will satisfy that design requirement. The lower half of Figure 1.5 shows the proposed model that will be explored in this work. Instead of directly connecting requirements to design objects, a new node called “Design Concept” will be embedded in the traceability link. Assembly of traceability links will be conducted by asking “what

concept should be applied to satisfy this requirement?” Solutions to this question establish links between requirements and design concepts. It is assumed that the design itself will correspond to the application of appropriate concepts. Thus, the links between design concepts and engineering objects represent actual implementations of concepts.

State-of-the-Art Traceability Model



Proposed Traceability Model

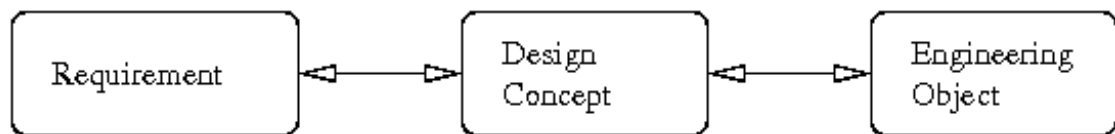


Figure 1.5: Simplified View of State-of-the-Art Traceability and the Proposed Model

From a validation and verification viewpoint, the key advantage of the proposed model is that software for “design rule checking” can be embedded inside the design concepts module. Thus, rather than waiting until the design has been fully specified, this model has the potential for detecting rule violations at the earliest possible moment. Moreover, if mechanisms can be created to dynamically load design concept modules into computer-based design environments, then rule checking can proceed even if the designer is not an expert in a particular domain.

From a modeling and visualization standpoint, this approach opens the door to improved methods for the visualization of requirements with respect to design

objects. In an ideal setting, the latter should be visualized using a notation easily recognized by the engineer (e.g., a mechanical engineering drawing).

Figure 1.5 is simplified in the sense that singular linkages are shown between the requirements, design concept, and engineering objects. As illustrated in Figures 1.3 and 1.4, a complete set of design requirements may assume a graph structure with multiple requirements mapping to multiple design objects (c.f., TRAMS). Figure 1.5 also implies that a design concept might be a singular module. In practice, detailed representations of problem domains will correspond to multiple design concepts that are related and constrained in a variety of ways. It is therefore anticipated that a “real-world” implementation will need to deal with ontologies and rule checking associated with ontologies. There needs to be a full understanding of the extent to which relationships and constraints in ontology-based descriptions of problem domains, working together with description logic reasoning engines, can influence and improve system-level design procedures, particularly in the early stages of development.

1.5 Smart Building Application

To better understand how the implementation of this vision might actually work in practice, consider development of a building architecture application, supplemented with modern support services.

Systems Engineering for a Smart Building with Modern Support Services.

Figure 1.6 shows an apartment floorplan plus a wireless network.

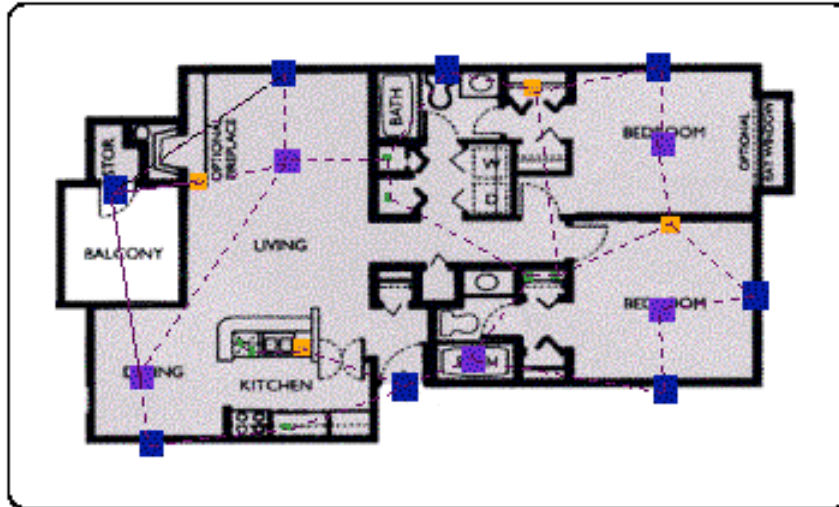


Figure 1.6: Modeling of a Wireless Network in a Smart Building

From a systems engineering perspective, this problem is appealing because an effective solution demands input and coordination of activities from multiple disciplines. Simplified approaches to this problem assume that the support services can be designed and installed after the building architecture issues have been settled. Certainly this is the case where new services are added to an existing structure. From an economic standpoint, however, the costs of designing and building structures are small in comparison to the costs of operating a building or other structure over the 10-30 or more years of its lifespan. Therefore, a key challenge for systems engineers is development of methodologies and tools, which allow the computer to play a pro-active role in the synthesis and checking of building architectures that need to support and interact with other engineering disciplines.

Established Approach to Architecture Development. Figures 1.7 and 1.8 show the established pathway of architectural floorplan development. During the

earliest stages of design, architectural concerns are directed toward development of functional requirements and identification of relevant design rules and economic and legal restrictions. The progressive transformation from required functional to constructive entities is very much a creative process. Initially, a systems architect may not know what types of components will be used for the design implementation – design development focuses on selection of components, and their preliminary position and connectivity to other components. System structures are created through the decomposition and clustering of spaces, followed by the progressive specification of geometric details. The symbolic layout level focuses on room contours, connected symbolic wall segments, and assignment of properties to regions. Simple geometry corresponds to thick walls, fleshed-out columns, and cut-out doors and windows. System behavior is enabled by the ability of the building occupants to function – the latter emanates from two sources: (1) functionality enabled by spaces and access to spaces, and (2) networked services (e.g., electrical, environmental micro-climates, security, etc.) integrated into the architectural domain. While many of these issues can be resolved with approximate/imprecise models of the final components to be used (de Vries, 2001), it is important to note that few opportunities exist to test the final product prior to its full implementation. Therefore, formal mechanisms that will enable early validation of designer intent and design rule checking can vastly improve the quality and reliability of the building system prior to deployment.

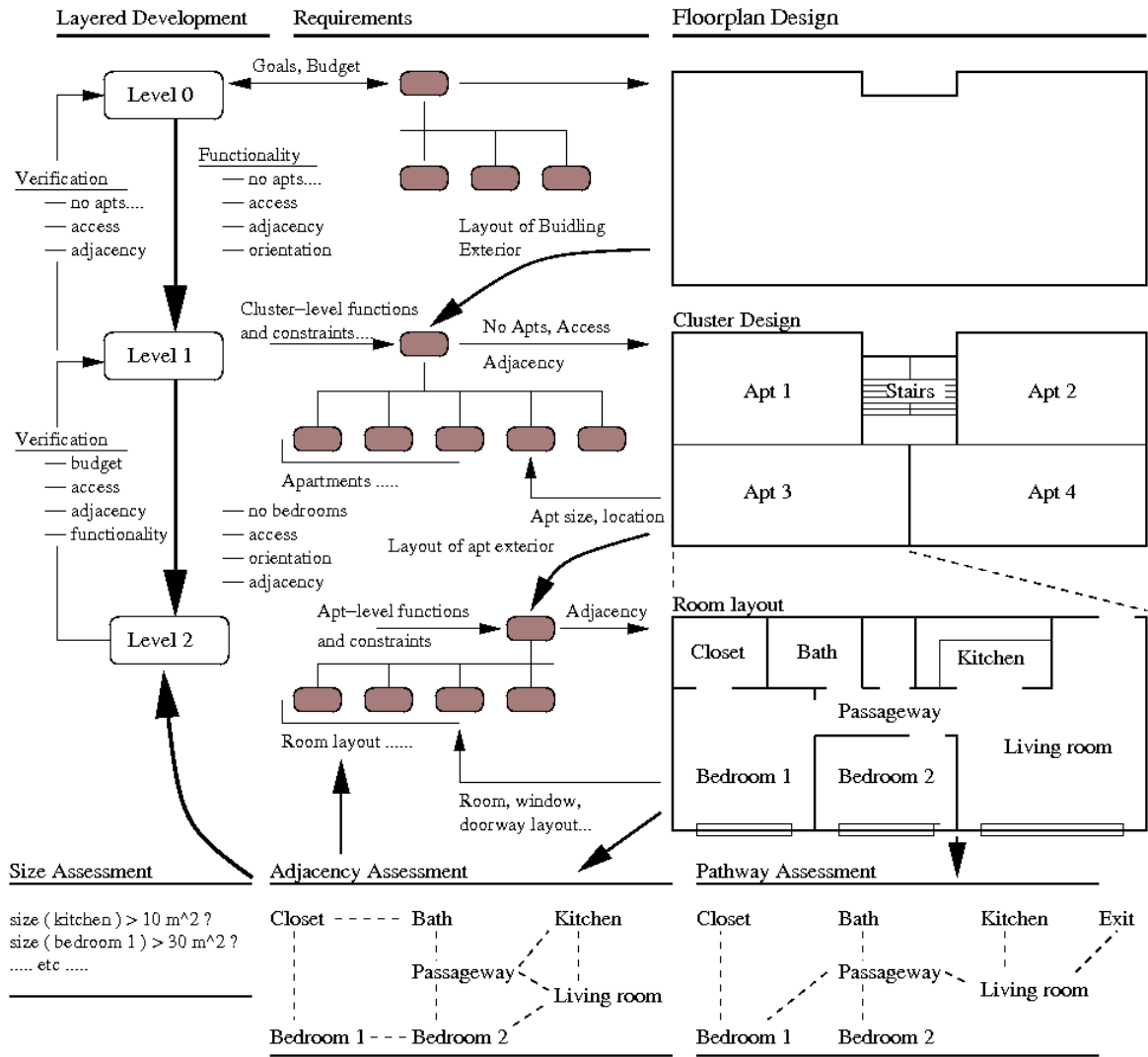


Figure 1.7: Progressive Decomposition of Architectural Floorplans

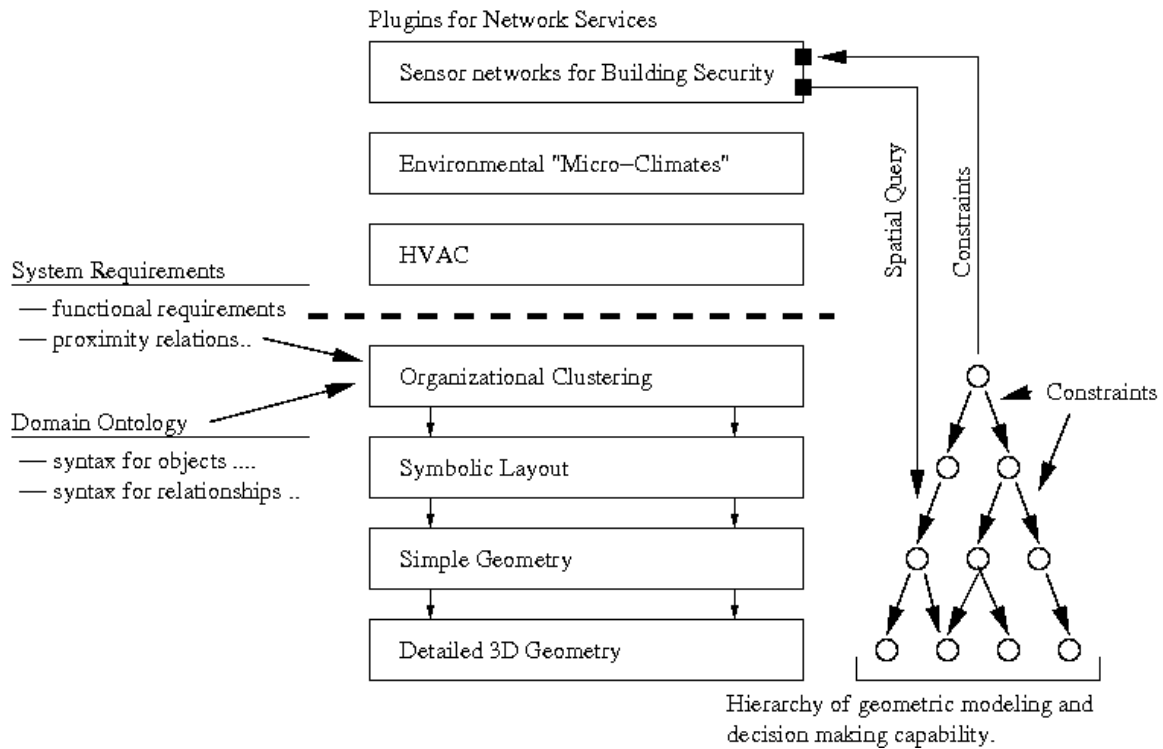


Figure 1.8: Framework for Multi-Level Development for Building Architectures (Spatial Arrangements) Augmented with Network Services (Adapted from Downs and Sequin (Downs 1999; Sequin, 1997))

Commercial tools such as AutoCAD (Madeen, 2001), 3D Home Architect and TurboCAD Deluxe/Professional (Broderbund 2004; Open, 2004) focus on the editing and presentation of Architectural/Mechanical CAD models/plans as blueprint-like drawings, 2-D designer viewpoints, and 3-D photorealistic renderings. Medium-end versions include support for pre-defined domain-specific features (e.g., architectural symbols), dynamic dimensioning, basic solid and surface modeling (i.e., boolean operations and slicing), collision detection, and cost estimation. High-end versions go even further, including support for sophisticated solid modeling, management of constrained design dependencies (d-cubed constraint technology), multi-monitor visualization and export to standard interchange formats (e.g., STEP).

Shortcomings of Present-Day Tools. From a systems engineering perspective, present-day tools for architectural development are limited in the following ways:

1. Support for separation of design concerns (e.g., from the beginning, topology/connectivity concerns are connected to geometry concerns) is weak.
2. There is a lack of comprehensive support for spatial reasoning. As such, the tools are not easily extensible to layers of services.
3. Support for traceability of requirements to the engineering system itself is nonexistent.

In defense of item 2, research tools have been created for the exchange of data/information associated with symbolic building design representations (Downs, 1999; Sequin, 1997) and to evaluate whether a building floor plan adheres to certain requirements and standards (Chun, 1997). Still, support for requirements traceability is completely missing.

Implementation of the Proposed Approach. Figure 1.9 shows the flow of data and information for the proposed approach. The development process begins with the layering and organization of requirements for various design concerns (e.g., behavior, structure, and testing), as shown in the top left-hand corner. If the building does not already exist, then these requirements will define the required system functionality. Constraints on performance and cost will define the design space.

In routine design, solutions are assembled from concepts pertinent to well-known design concerns, contexts, and viewpoints (Rosenman, 1996). UML can

play a central role in the representation and visualization of intermediate products (i.e., application of “principles and practices” employed by professionals). For example, UML class and object diagrams are an ideal representation for: (1) Concepts (and relationships among concepts) associated with a particular problem domain or design concern, and (2) Organization (clustering and decomposition) of spaces into hierarchies.

The dashed arrows connecting requirements to UML classes are established by asking the question: What concept will be applied to satisfy this requirement? Then once that link is established, the continuing link to the engineering model is easily established – it is, after all, just the object instantiation of concepts modeled in the UML class diagram. On the back end, each class will contain attributes and methods needed to quantitatively evaluate object instances. Some of this information may not be explicitly visible (e.g., exact square footage of a region, function of room, designated occupant, etc.). This extra information should be readily retrievable with a simple mouse or menu action. Research is needed to better understand the extent to which various types of constraints are supported by the UML class viewpoint.

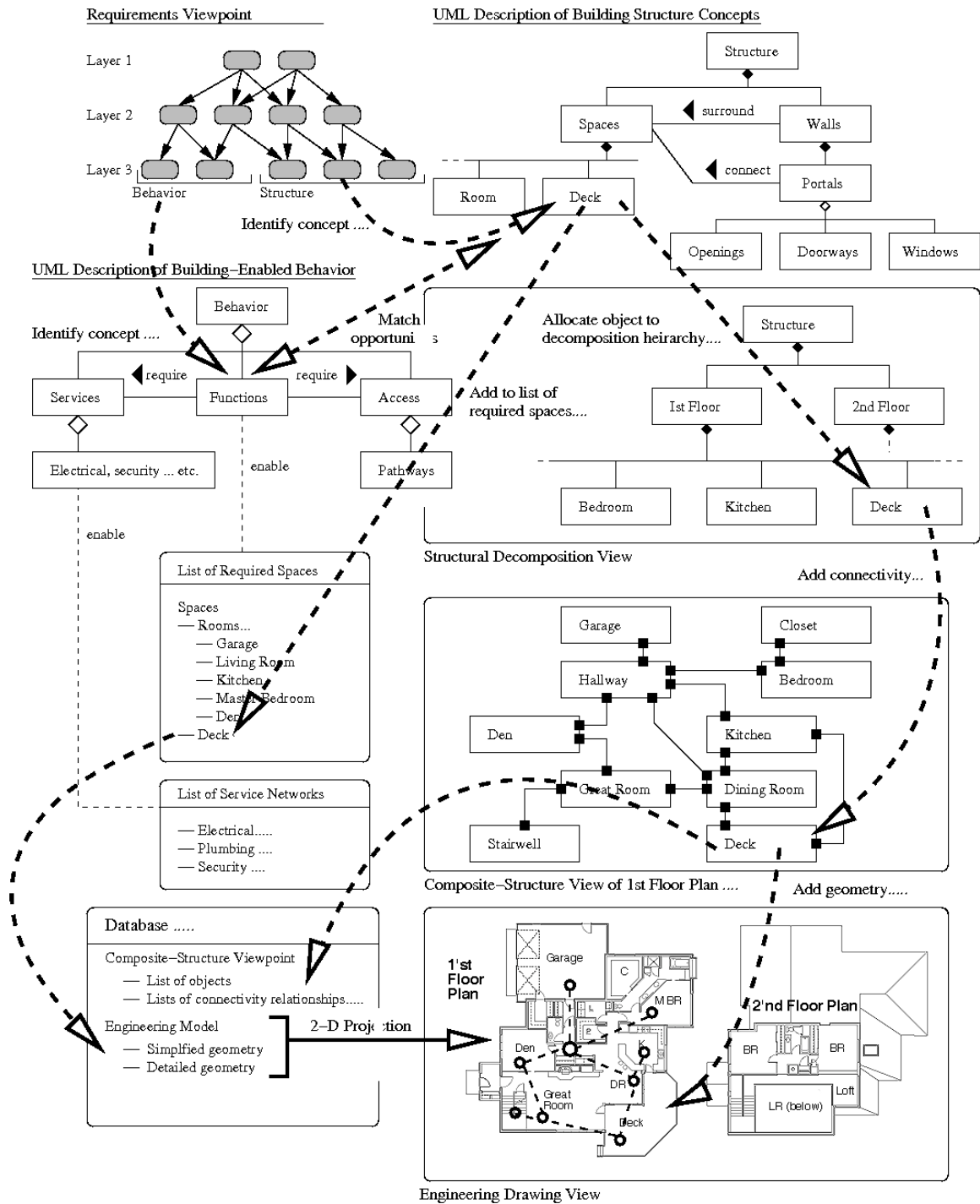


Figure 1.9: Pathways from Requirements to UML Representations of System Behavior/Structure Concepts to Multi-Level Representations of Building Architecture. (Source: (Austin, 2005))

System-level design alternatives are created by linking models of system-level behavior to the high-level structure, and imposing constraints on performance and

operation (e.g., control logic; temporal logic; spatial logic). Floor planning processes need to adhere to three types of constraints: (1) topological (i.e., orientation, traffic/pathway, and location/adjacency concerns), (2) dimensional (i.e., size and space concerns) and (3) functional (e.g., aesthetic concerns) (Cao, 1990). If the required functionality at lower levels of development cannot be satisfied (perhaps because the constraint values are too stringent), then the verification process will fail and the high-level developments will need to be adjusted to accommodate the demands of the lower level requirements (e.g., perhaps a space would need to be increased in size). The factoring process is guided by functionality that the region is expected to provide, and restricted by topological/geometric constraints (Kharrufa, 1985; Madeen, 2001).

The heavy dashed arrows in Figure 1.9 represent traceability links connecting requirements to specific system-level design concepts, which, in turn, will be implemented as entities in a building architecture object-model. Looking forward, designers should be able to click on a requirement and trace its implementation through the concept, structural decomposition, composite-structure and engineering drawing models. Conversely, designers should be able to click on an object (or group of objects) in a drawing and trace its existence back to a specific requirement (or groups of requirements). In this scenario, a drawing is a detailed two-dimensional projection of an engineering model.

1.6 Ontologies and Ontology-Enabled Computing

An ontology is a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic (Gomez-Perez, 2004; Hendler, 2001; Staab, 2000). Ontologies are needed to facilitate communication among people, among machines, and between humans and machines.

Instead of creating a system through the integration of data, systems are created through the application and integration of concepts. System and sub-system evaluation will depend on both the concept and the data used in its implementation (e.g., an area constraint will depend on geometry).

To ensure that system-level designs are faithful representations of both the stakeholder needs and the capabilities of the participating application domain(s), ontology models need to possess several attributes (Shanks, 2003):

Accuracy. The system-level model needs to accurately represent the semantics of the participating application domains, as perceived by the project stakeholders.

Completeness. The system-level model should completely represent the relevant semantics of the problem domain, as perceived by the project stakeholders.

Conflict-free. The semantics used in various parts of the system-level model and/or various application domains should not contradict one another.

No redundancy. To reduce the likelihood of conflicts arising if and when the model is updated, the model should not contain redundant semantics.

Ontologies may be built using a variety of representations and languages. Representations can be highly informal (e.g., natural language), semi-informal (i.e., they have a formally defined language), and rigorously formal (i.e., formal semantics, theorems, and proofs of properties). The ontology community distinguishes ontologies that are mainly taxonomies from ontologies that model domains more in depth, applying restrictions on domain semantics (Gomez-Perez, 2004). So-called *lightweight ontologies* include concepts, concept taxonomies, relationships between concepts, and properties of the concepts. So-called *heavyweight ontologies* add axioms to lightweight ontologies – axioms serve the purpose of adding clarity to the meaning of terms in the ontology. They can be modeled with first-order logic.

Top-level ontologies describe general concepts (e.g., space, connectivity, etc.). Domain ontologies describe a vocabulary related to a particular domain (e.g., building architecture, plumbing, etc.). Task ontologies describe a task or activity. Application ontologies describe concepts that depend on both a specific domain and task. These ontologies might represent user needs with respect to a specific application. Because a unified theory for system validation does not exist at this time, present-day procedures for design rule checking tend to focus on small snippets of the system model functionality and are achieved in several ways: (1) consistency checking, (2) connectivity analysis, and (3) model analysis on a global basis, based upon graph-theoretic techniques.

Ontology-Enabled Computing. To provide for a formal conceptualization within a particular domain, and for computers to share, exchange, and translate

information within a domain of discourse, an ontology needs to accomplish three things (Liang, 2004):

1. Provide a semantic representation of each entity and its relationships to other entities;
2. Provide constraints and rules that permit reasoning within the ontology;
3. Describe behavior associated with stated or inferred facts.

Items 1 and 2 cover the concepts and relations that are essential to describing a problem domain. Items 2 and 3 cover the axioms that are often associated with an ontology. Usually, axioms will be encoded in some form of first-order logic.

The Semantic Web. This project assumes that advances in ontology-enabled design and development will occur in parallel with advances in the Semantic Web. In his original vision for the World Wide Web, Tim Berners-Lee described two key objectives: (1) To make the Web a collaborative medium; and (2) To make the Web understandable and, thus, processable by machines. During the past decade the first part of this vision has come to pass – today’s Web provides a medium for presentation of data/content to humans. Machines are used primarily to retrieve and render information. Humans are expected to interpret and understand the meaning of the content. The Semantic Web (Berners-Lee, 2001; Hendler, 2001) aims to give information a well-defined meaning, thereby creating a pathway for machine-to-machine communication and automated services based on descriptions of semantics (Geroimenko, 2003). Realization of this goal will require mechanisms (i.e., markup languages) that will enable the introduction, coordination, and sharing of the formal semantics of data, as well as an ability to

reason and draw conclusions (i.e., inference) from semantic data obtained by following hyperlinks to definitions of problem domains (i.e., so-called ontologies).

Figure 1.10 describes the infrastructure that will support this vision (Berners-Lee, 2000). The Resource Description Framework (RDF) defines a standard for describing the relationships between objects and classes in a general but simple way. Class relationships and statements about a problem domain are expressed in DAML+OIL (DARPA Agent Markup Language) and more recently, the Web Ontology Language (OWL) (Web, 2003).

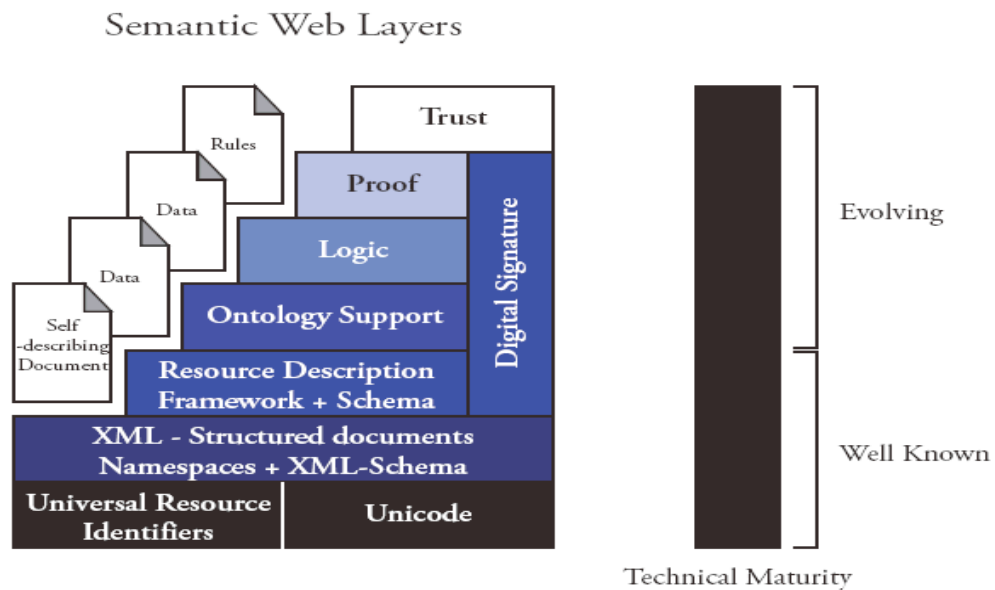
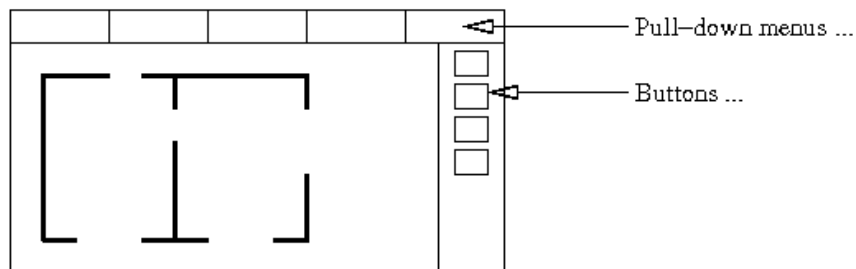


Figure 1.10: Semantic Web Layer Cake Technologies and their Maturity

1.7 Framework for Interactive Ontology-Enabled Design

Graphical Interface. Figure 1.11 contrasts present-day graphical user interfaces for engineering design versus the proposed method. Present-day environments for design use settings on pull-down menus and button components to control the context of an editing operation. Typically, a designer will enter a mode, make an adjustment to the design, and then exit the mode. Notice that the design environment supports neither an explicit representation for the system requirements, nor an explicit representation of concepts/concerns employed in the design. Techniques for integrating projections remain more of an art than science.

Present-day Graphical Interfaces



Proposed Graphical Interface

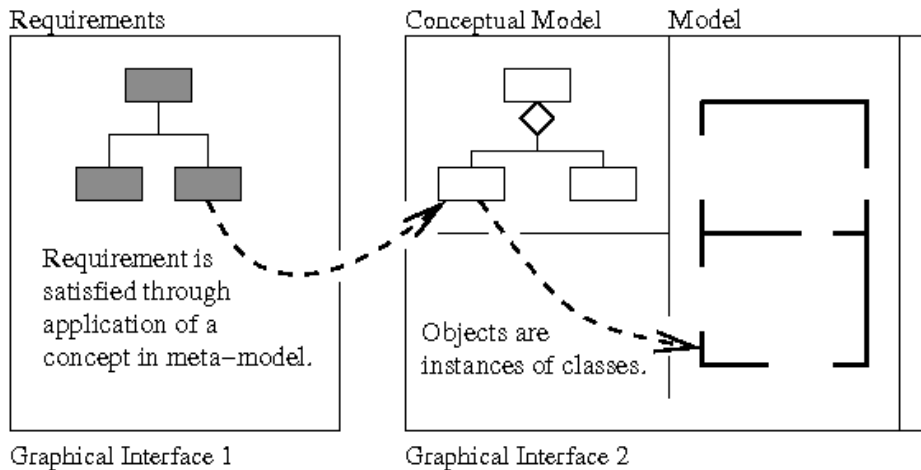


Figure 1.11: Present-day and Proposed Graphical User Interfaces

Here, in contrast, it is proposed that selection of a specific ontology be used as the means for creating new elements in a design. Each symbol in the conceptual model will have a concrete executable definition. UML class diagrams will correspond to a graph of interacting “concept elements” that may be used to generate many potentially effective design prototypes. Changing the value of a parameter in a particular design element may affect its feasibility – however, it will not affect the underlying conceptual model. In all cases, the conceptual model will enable services for traceability and design rule checking.

Viewpoint Integration. The lower half of Figure 1.11 is simplified in the sense that it implies one design concept will lead to one engineering object. In practice, most engineering entities serve multiple functions, which in turn implies that they contribute to the satisfaction of multiple design viewpoints (or design concerns).

In order for multiple viewpoint model-based development to proceed in a disciplined way, there needs to be formal models describing “interactions” and “restrictions” among the various viewpoints. Horizontal integration joins together abstractions for separate concerns (e.g., functional, structure, interface, etc.). Vertical integration links viewpoints together for concerns represented at different levels of abstraction. This task is complicated by the fact that each discipline will interpret design objects relevant to its set of concerns. Moreover, as shown in Figure 1.12, different disciplines may not use the same terms to describe the same design object. (e.g., an architect will refer to a horizontal plane as a floor; a construction engineer might call the same object a slab).

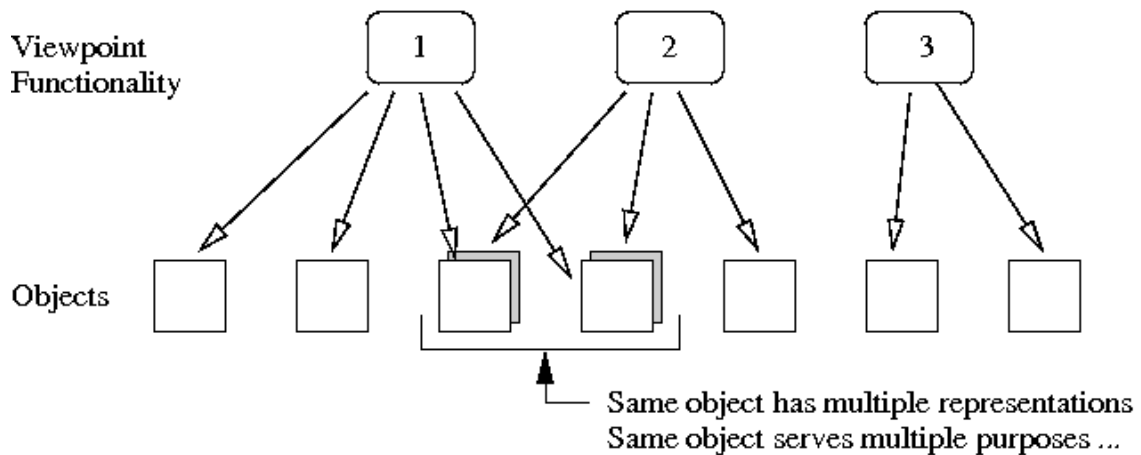


Figure 1.12: Multiple Conceptual Interpretations of a Single Design Object

Recognition of interactions among viewpoints is the first step toward integration of entities in different abstractions. Figure 1.13 shows, for example, viewpoints 1 and 2 containing entities 1 and 2, respectively. In UML terminology, the “interaction mechanism box and associated dashed line” is an association relationship. Its purpose is to describe an explicit relationship between entities 1 and 2. Examples include: (1) same as (i.e., the element has all of the properties of the “named” element), (2) element of (i.e., the element is a component of the “named” element), (3) part of (i.e., the element forms part of the “named” element), and constrained by (i.e., a property of an element is constrained by the property of another).

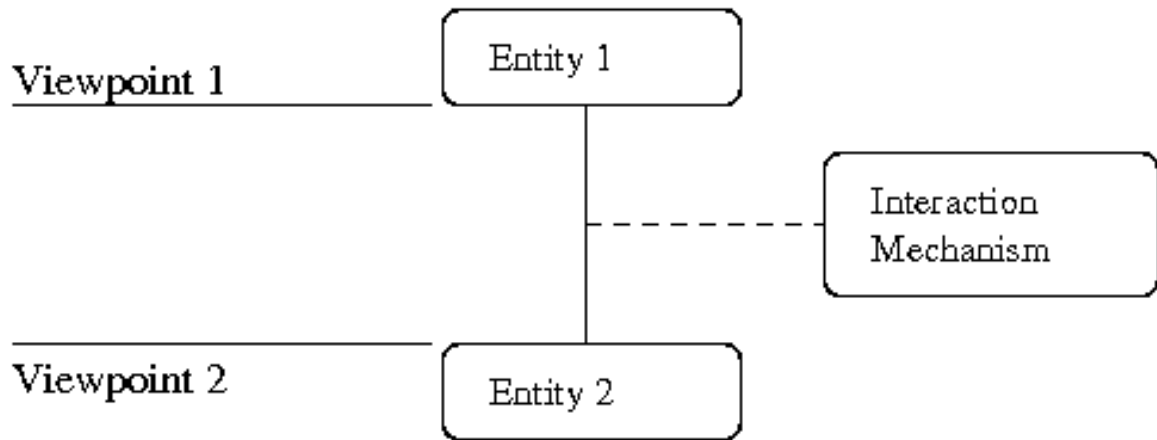


Figure 1.13: Schematic of Viewpoint Interaction

1.8 Project Scope and Objectives

The scope and objectives of this project will be limited to the implementation and evaluation of a software prototype. Unfortunately, the “building architecture with support services” application is too difficult for a first cut implementation. Instead, this project will focus on a simpler example: Modeling and Visualization of the Washington, D.C. Metro System. The first-cut implementation will:

1. Focus on early stages of design, where component selection, positioning and connectivity are the principle concerns.
2. Represent ontologies as UML class diagrams.
3. Not consider system- and component-level behavior.
4. Not consider assignment of functions to components.

The expected benefits are as follows:

1. Provides a direct link from a requirement (or design concept) to its implementation in the engineering design.
2. Looking back from the design, builds the rationale for the implementation right into the traceability model.

3. Activates the rule checking associated with implementation of design concepts (i.e., classes in UML diagrams) at the earliest possible point in the design process.

The remainder of this thesis describes the conceptual framework needed for the first-cut implementation. Chapter 2 describes conceptual modeling with UML. Chapter 3 discusses the software architecture design using listener-driven events and the Violet UML Editor. Chapter 4 describes the Washington, D.C. Metro System software model. Conclusions and future work are covered in Chapter 5.

Chapter 2 Design Concept Modeling with UML

The purpose of this chapter is to create a framework for design concept modeling with UML. This framework is a prerequisite to the development of tools that will support the early, and possibly real-time, validation/verification of design implementations.

2.1 Development of System-Level Design Representations

At the system level, designs are viewed as collections (i.e., networks and hierarchies) of large, arbitrarily complex functional units that form the major components of a system. Designers need to identify components/objects, their attributes and functions, and interfaces and relationships to external entities. Analysis should emphasize the clear and concise definition of system interfaces and hide internal details of module implementations.

Starting with a coarse and fragmented representation of a system, architecting is all about selection, positioning, and connectivity of appropriate components. Sometimes appropriate problem domain concepts are organized into class hierarchies. Transformations can refine the interaction among components or decompose the system description into finer levels of detail, as shown on the right-hand side of Figure 2.1. Challenges in design include: (1) A multiplicity of good answers or abstractions, and (2) A lack of an acid test for determining when an abstraction has captured all of the relevant details from a “real-world” situation. Still, before a designer proceeds with detailed design and

implementation, it is critically important that he/she has an up-front design abstraction that is right (Barker, 2005).

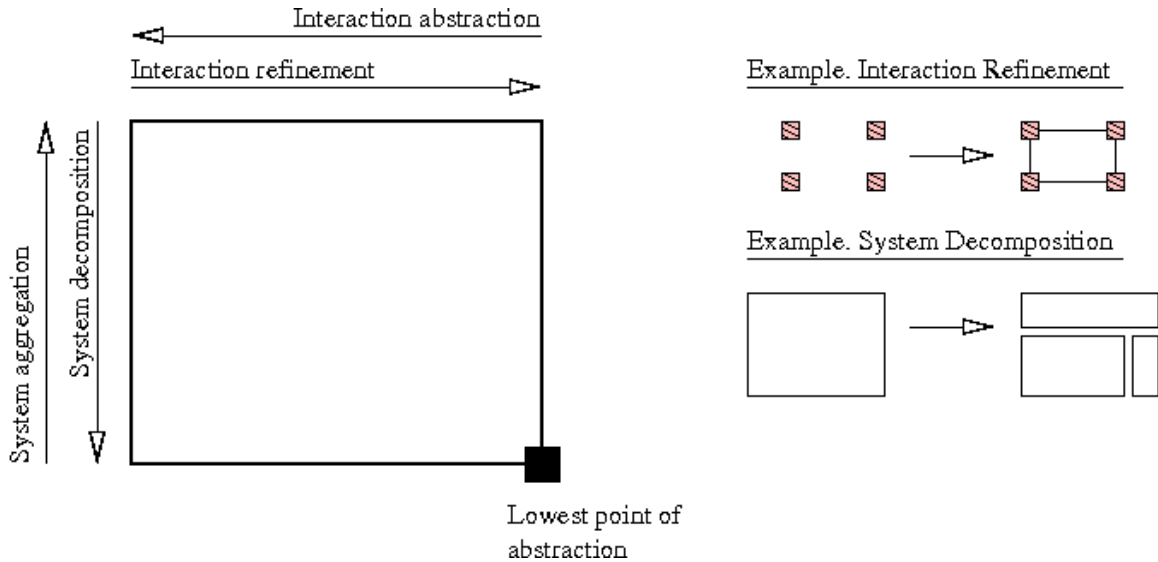
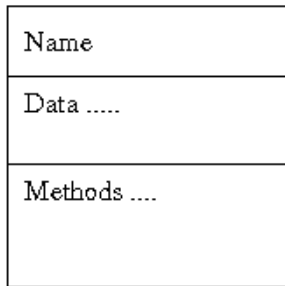


Figure 2.1: Magic Square of System Development (Wieringa, 1998)

Classes and Objects. It is assumed that system applications will be assembled from objects – in other words, objects are the fundamental building blocks. In software applications and as illustrated in Figure 2.2, the state of an object is defined by its data. Behavior is defined via the methods that can manipulate the data and communicate with other objects. Of course software objects can only exist if certain rules of operation are satisfied – these rules are embedded in the syntax/semantics of the language in which the software is being written. For system-level development, objects will have state, behavior, and rules under which they will operate.

Classes for Software



Classes for System-Level Design

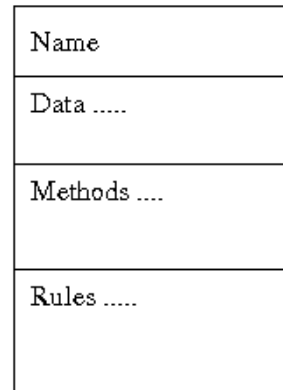
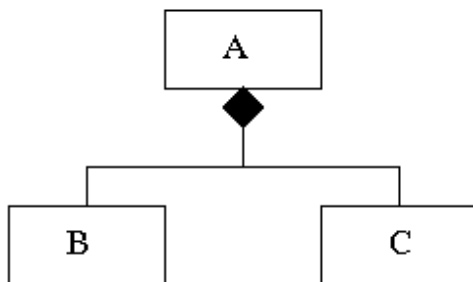


Figure 2.2: Definition of Classes for Software and System-Level Design

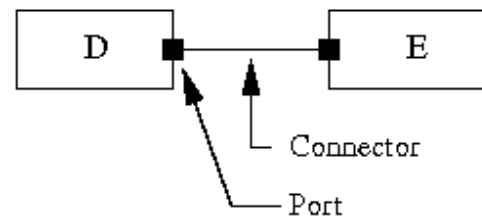
Hierarchy and Network Abstractions for Component/Object Assemblies. The EIA-632 (Martin, 2003) standard assumes that the basic systems engineering unit is a product or process block. Models for complicated systems engineering products and processes correspond to hierarchies and/or networks of simpler products/processes. Simple examples of each type are shown in Figure 2.3.

Hierarchy Abstraction



Module A contains modules B and C

Network Abstraction



Module D is connected to module E

Figure 2.3: Hierarchy and Network Abstractions for SE Development

Connections among units may be arbitrarily complex, carrying unspecified data and information (all that matters is whether or not a particular function block connects to another). Figure 2.4 shows a simple system hierarchy decomposed

into two levels, with a network of components (B and C) at level 2. Data values at lower levels aggregate into the data values at higher levels. The levels in a dimension are organized into one or more hierarchies.

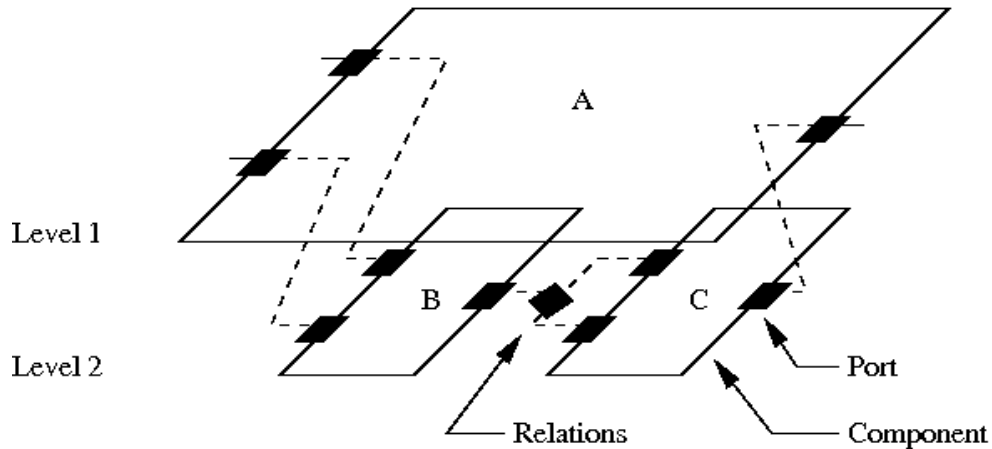


Figure 2.4: Hierarchy and Network Abstractions for SE Development

The system structure must satisfy the following constraints:

1. Within a hierarchy, each level is logically connected to the levels above and below it.
2. A port cannot be contained by more than one entity.
3. Links cannot cross levels in the hierarchy,
4. Port-to-port communications must have compatible data types (e.g., signal, energy, force, etc.).

Evaluation mechanisms should provide the designer with critical feedback on the feasibility and correctness of the system architecture.

2.2 Representing Ontologies with UML Class Diagrams

From a systems engineering perspective, the key advantage in modeling design concepts with Semantic Web languages such as RDF, DAML and OWL, is that software tools have been developed for logical reasoning with relationships and rules implied by ontologies and for evaluation of assertions (see Figure 1.10). Unfortunately, at this time RDF, DAML and OWL lack a standard representation for visualizing concepts expressed in these languages.

A practical way of overcoming this shortcoming (at least for the meantime) is to use UML class diagrams – actually, graph structures of UML schema – in lieu of a formal ontology. UML is well defined and has a community of millions of users. UML class diagrams can be used for representing concepts (and their attributes), and relations between concepts (e.g., knowledge reflecting performance and legal and economic restrictions). Basic relationships, such as inheritance and association, can be modeled. Axioms (i.e., additional constraints) can be represented in the Object Constraint Language (OCL).

This idea is not new. The close similarity of DAML and UML has been established by Cranefield and co-workers (Cranefield, 2001b; Cranefield, 2001a; Guizzardi, 2004). For example, both DAML and UML have a notion of a class which can have instances. The DAML notion of a `subClassOf` is essentially the same as the UML notion of specialization/generalization. Thus, UML qualifies as a visual representation for ontologies (Baclawski, 2001). Moreover, tools are starting to emerge for the automated transformation of ontologies to UML. See, for example, descriptions of the tool DUET in Kogut et al. (Kogut).

Meta-Models and Meta-Meta-Models. Most engineers think of UML as simply a diagramming notation for the high-level, albeit informal, specification of system structure and behavior. UML is, in fact, based on well-defined language concepts specified in terms of meta-models and meta-meta-models. Diagrams are one representation of the UML language concepts. An equivalent XML representation also exists.

A meta-model describes information about models. Meta-meta-models describe information about meta-models. Figure 2.5 shows the pathway from meta-meta-models to meta-models to models and implementation of engineering systems.

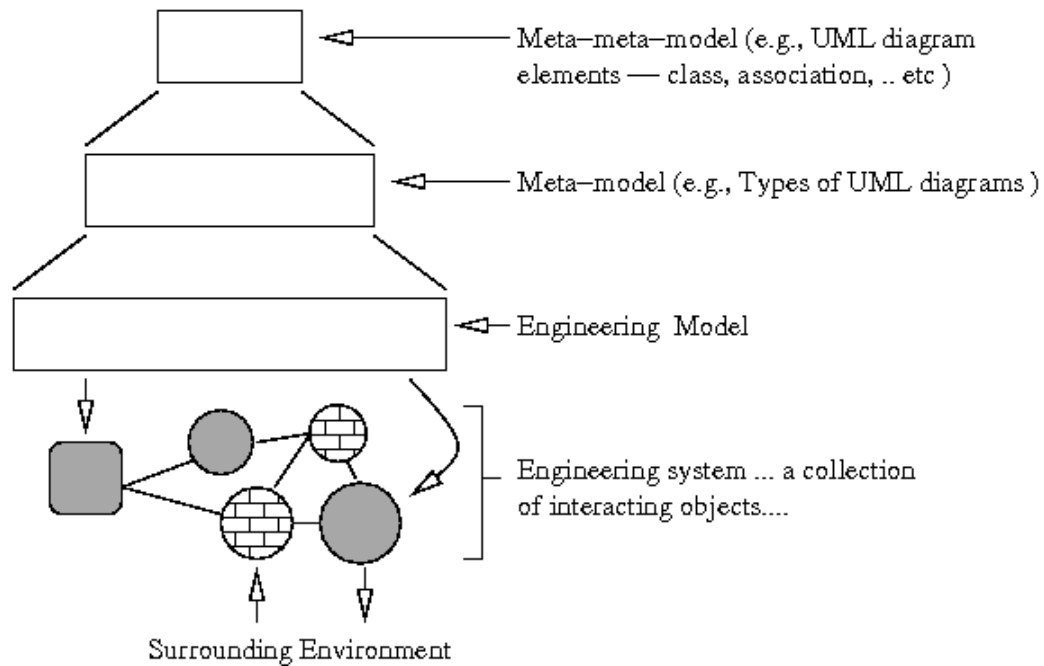


Figure 2.5: Pathway from Meta-meta-models to Engineering Models and Systems (Wieringa, 1998)

Key points:

1. The meta-meta-model (also known as the UML meta-model) is a model that describes the UML language – specifically, it describes classes, attributes, associations, packages, collaborations, use cases, actors, messages, states, and all the other concepts in the UML language.
2. UML-like diagrams express concepts and relationships among concepts suitable for creating a design. These diagrams serve as a meta-model for the development of potentially acceptable designs.
3. The UML diagrams themselves are created from diagram elements, which have well-defined semantic meaning. The set of diagram elements (e.g., notations for inheritance, aggregation, and so forth) form a meta-meta model.
4. Requirements are satisfied by applying a concept expressed in the meta-model. The activation of a concept results in an object in the design model.

The latter is shown on the bottom right-hand side of Figure 2.6.

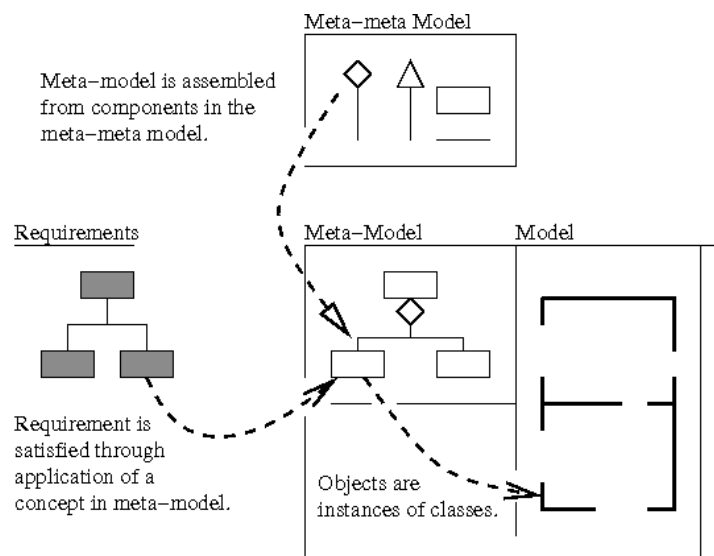


Figure 2.6: Roles of the Meta-Meta-Model and Meta-Model in the System Assembly

2.3 Relationships between Classes

Class diagrams model the class structure and contents of an application through a combination of objects, classes and packages. By themselves, individual classes and their corresponding object implementations do not constitute a system. Systems are created from groups of related classes that interact in some manner. It follows that in order to understand a system, there needs to be an understanding of the types of relationships that can exist between classes.

Links and Associations. Links and associations establish relationships among entities within the problem world or the solution world. As illustrated in Figure 2.7:

1. Associations are descriptions of potential links with a common implementation. Put another way, an association specifies how an object type can be specified in terms of other object types.
2. Links are structural relationships between two or more objects. They are instances of associations.

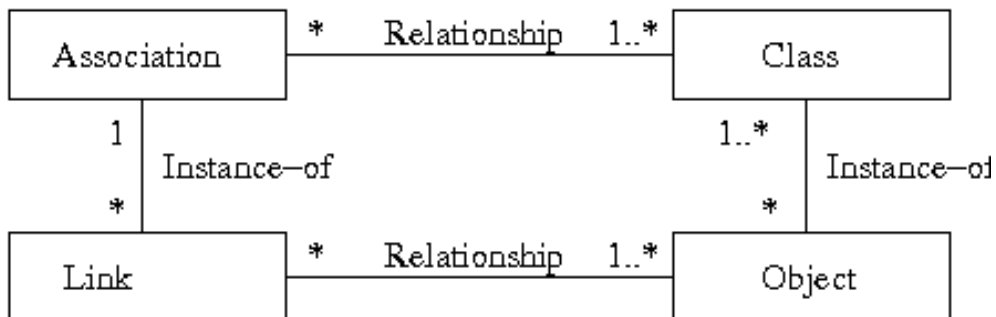


Figure 2.7: Meta-model for Association and Link Relationships

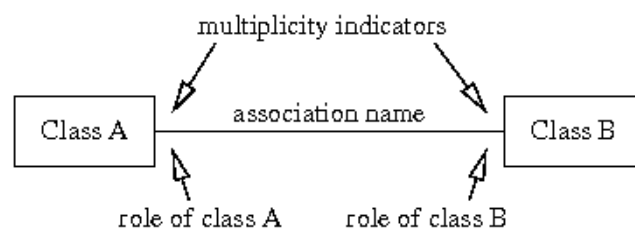
Most relationship issues among classes can be resolved through the answer to three questions: (1) “uses a,” (2) “has a,” and (3) “is a.” From these classifications, complex phenomena can be systematically described via networks and hierarchies of classes.

Association (“uses a”). The “uses a” relationship expresses a situation where one class may communicate with a second class, and employ its data and services.

Binary associations express static bi-directional relationships between two classes. The upper half of Figure 2.8 shows the meta-model for a binary association – the solid line connects the participating classes. Labels indicate the role of the association.

The lower half of Figure 2.8 shows a binary association between classes A and B. The label “inside” expresses the form of this relationship. The example does not specify a multiplicity relationship. Association relationships can also be represented as a separate class, as illustrated in Figure 2.9.

Meta-model for Binary Associations



Example

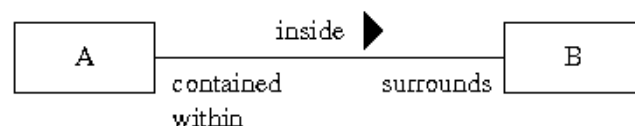


Figure 2.8: Binary Associations in UML: Meta-Model and Example (Barker, 2005)

Binary Association

Association Class

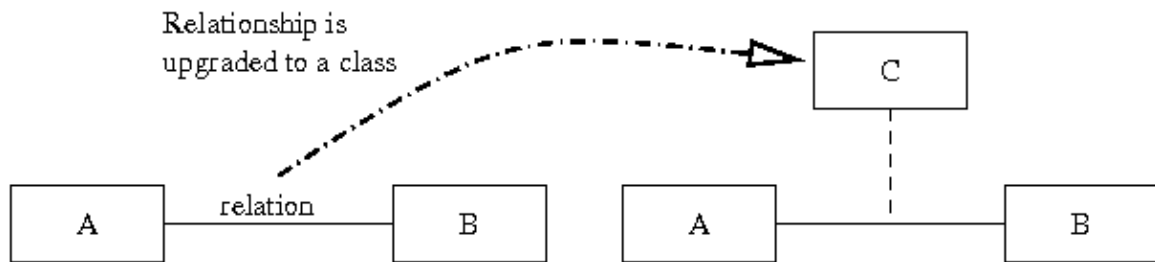


Figure 2.9: Association Relationships in UML

Multiplicity. Multiplicity relationships determine the minimum and maximum instances of a class that are allowed in an engineering/object implementation. Therefore, multiplicity relationships apply to associations. Links between objects always occur in a pairwise fashion. Figure 2.10 summarizes the multiplicity relationships used in UML. Three types of relationships need to be considered:

- 1. One-to-One (1:1).** One instance of class A is related to one instance of another class, say class B.
- 2. One-to-Many (1:M).** One instance of class A is related to many instances of class B.
- 3. Many-to-Many (M:N).** Many instances of class A are related to many instances of class B and vice versa.

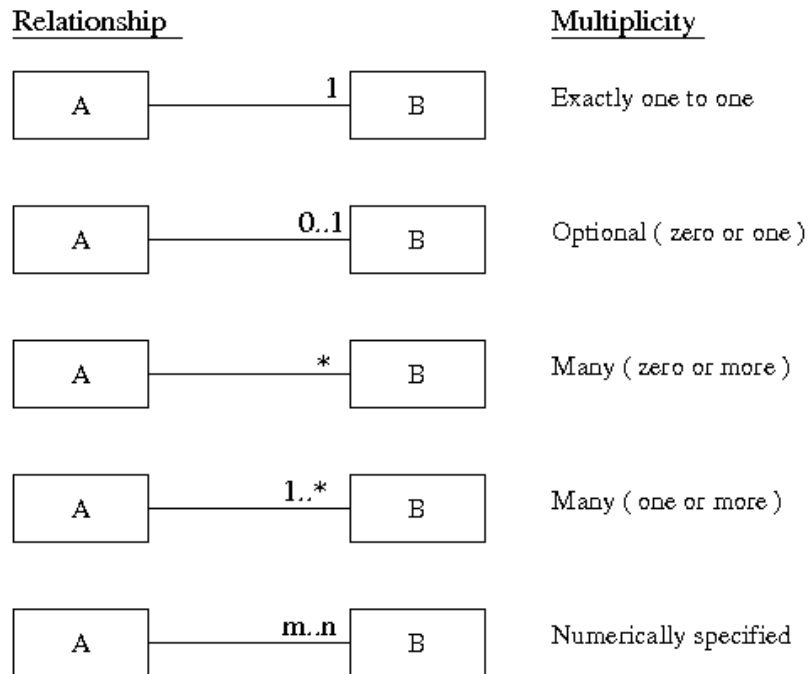


Figure 2.10: Multiplicity Relationships in UML

Of course, the validity of a particular type of multiplicity will depend on the system features being modeled.

Aggregation and Composition (“has a”, a.k.a. part-whole). Aggregation and composition are special forms of association. The “has a” question determines aggregation. For example, the aggregation relationship in Figure 2.11 reads “an instance of class A has an instance of classes B and C.”

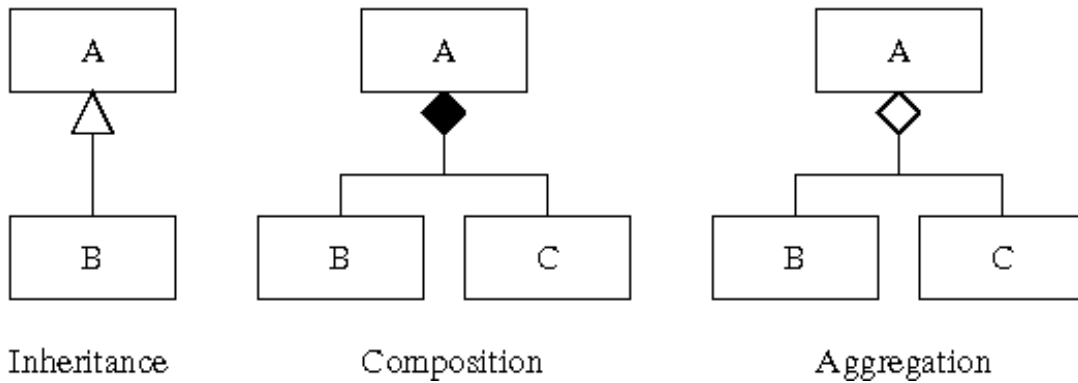


Figure 2.11: Inheritance, Composition and Aggregation Relationships in UML

The entity forming the aggregation is the whole. The objects being aggregated are the parts. An appropriate declaration in Java is:

```
public class A {
    B bRef; // reference to an object of type B.
    C bRef; // reference to an object of type C.
}
```

Composition is a special form of aggregation that occurs when an object aggregates another object, but also (for some reason) has control over that object. The control can manifest itself in a number of ways. For software applications, control can mean that both objects are constrained to have the same life-cycle (i.e., when the first object dies, the second object dies too). For hardware applications, control can mean when the object moves, all of the dependents will move too.

Inheritance (“is a”). Organizing classes into an inheritance relationship is relevant when a group of classes have similarities, but also some differences. The similarities among classes are bundled into a general (or generalized) class. The distinguishing features are represented in classes that inherit features from the “generalized class” but also add new features. The inheritance relation in Figure

2.11 says “B is a specialized version (or a type) of A.” The appropriate declaration in Java is:

```
public class B extends A { ... }
```

2.4 Human-Ontology Interaction and Processing

Standard implementations of computational support for UML diagramming have the goal of providing end-users with the ability to easily create static diagrams. Here, in contrast, UML classes and class diagrams serve the dual role of: (1) representing domain ontologies, and (2) enabling linkages between requirements and engineering objects. Computational support has the goal of providing executable services for design traceability and design rule checking.

Implementation. Figures 2.12 and 2.13 show the step-by-step procedure for development, implementation and operation of ontologies in a design specification setting. The implementation needs to support:

1. Definition of relationships (e.g., one-to-one, one-to-many, etc.).
2. Management of relationships (e.g., create, trace, and remove).
3. Inquiry for availability of services.

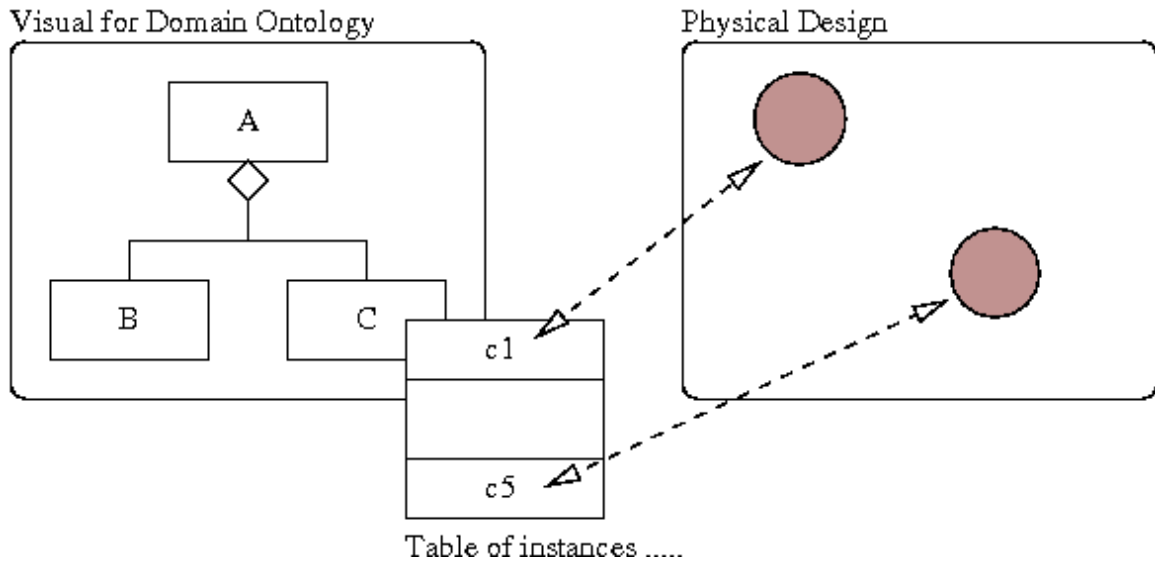


Figure 2.12: Connectivity between the Specification and Physical Models

Looking forward, and as shown in Figure 2.12, each specification class will store tables of references to objects in the physical design. Looking backward (not shown), these references will be connected to one or more design requirements. Figure 2.13 shows the pathway of development for processing of events – the exact details of how this will work remain to be worked out.

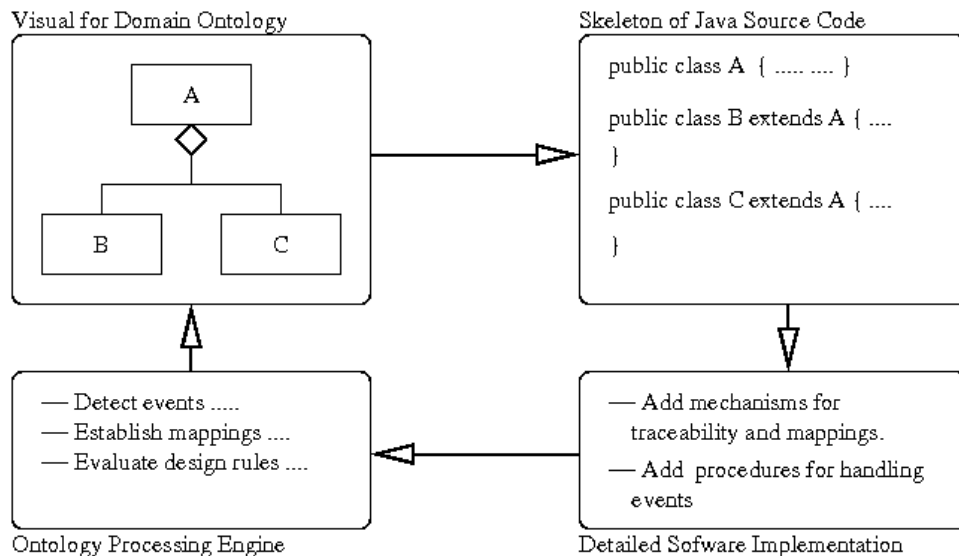


Figure 2.13: Step-by-Step Implementation of Ontology Processing Machine

Chapter 3 Software Architecture Design

3.1 Software Architecture Design

The software architecture design is concerned with the selection and configuration of major software components and their relationships. For this problem domain, and as illustrated in Figure 3.1, the software architecture is required to allow for the linking of requirements to UML classes to object representations in the engineering model. In start-of-the-art implementations, each of these models will be autonomous with high/strong cohesion. Cohesion is the degree of connectivity among elements in a single module. High levels of cohesion occur when a module performs a small number of related and well-defined functions (e.g., a software tool for management and visualization of design requirements).

As illustrated in Figure 3.1, the traceability mechanisms are required to link the models together in a manner that is precise, yet minimizes coupling between adjacent systems. In software, coupling is the degree to which each program module relies on, or associates with, another module – it is the single greatest problem in large software systems (Faison, 2006). Weak coupling implies that one module does not have to be concerned with another module's internal implementation. Instead, interacting is done with a stable interface (Coupling, 2006). Weak coupling will facilitate the ease with which individual traceability links may be added/removed and, in the longer term, entire versions of the requirements/ engineering system model may be swapped out and upgraded.

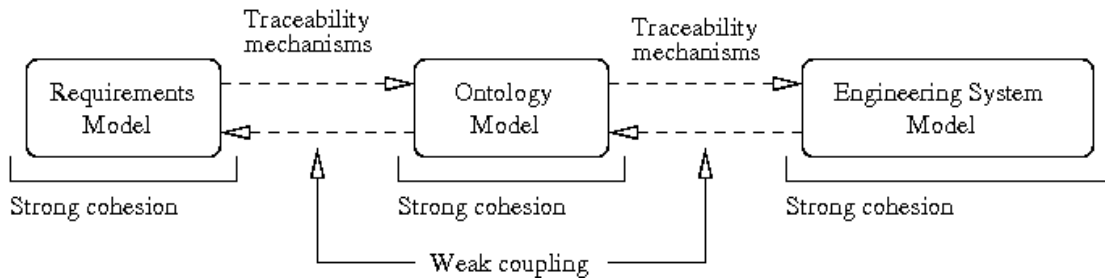


Figure 3.1: Overview of System Architecture Requirements

The working system must be capable of detecting and handling relevant external events, and notifying a designer when the design specification becomes incompatible with the ontology requirements. A fully operational system will need to support the representation and visualization of many requirements and ontologies. For the prototype implementation, however, it is assumed that the number of requirements will be small (i.e., less than 10) and that all of the requirements can be connected to a single ontology. This simplification will allow the software development to focus on the creation and operation of traceability links.

3.2 Graphical User Interface (GUI) Design and Traceability Support

Figure 3.2 shows the proposed GUI layout. There are three main panels – a requirements panel containing the table of requirements, a UML diagram panel containing the class diagram of the system, and an engineering model panel containing the model of the system.

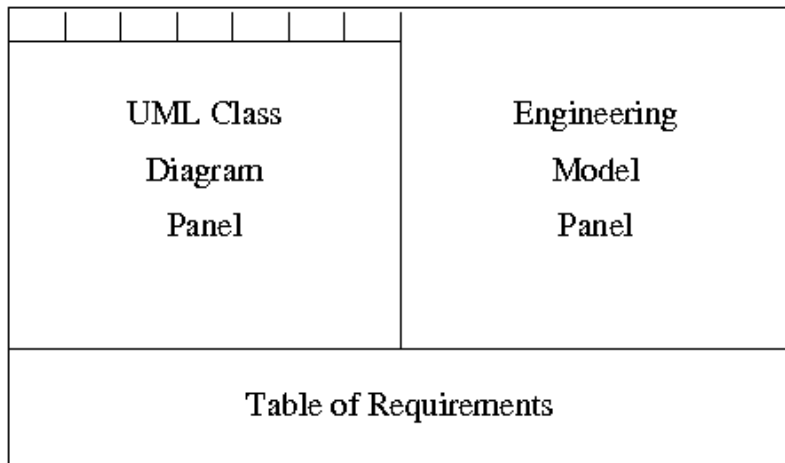


Figure 3.2: GUI Layout

XML is used to represent the data stored in each of the panels. Figure 3.3 shows the connections of the XML representation to the GUI. Each of the three panels parses its corresponding XML file, retrieves the necessary data from the XML file, and displays the data on the panel as a specific internal representation.

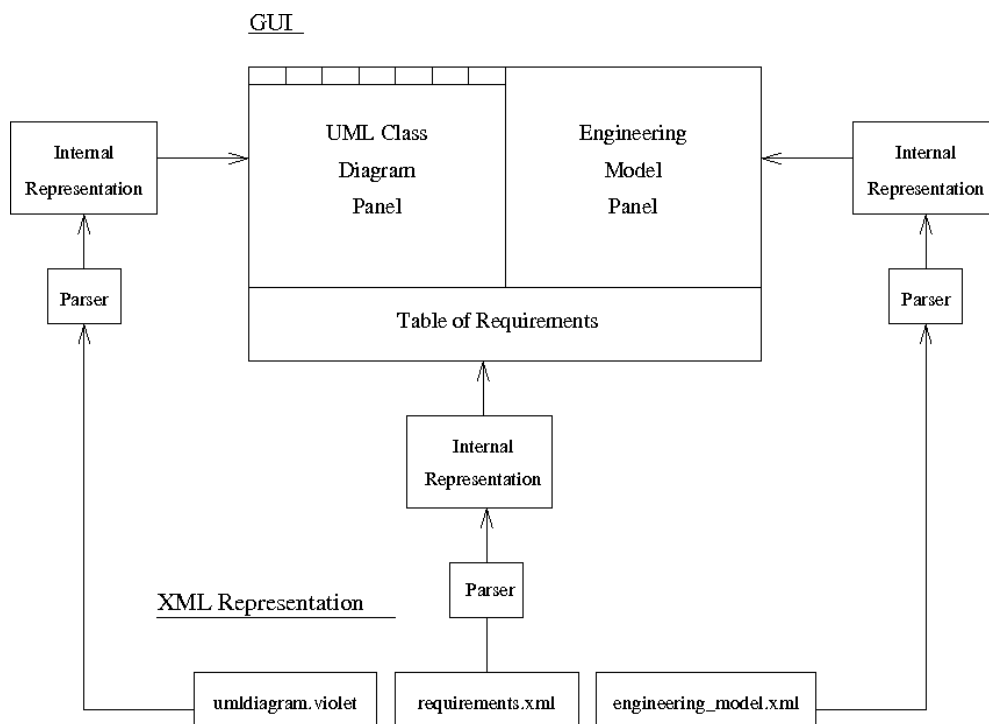


Figure 3.3: Connection of XML to the GUI

Once these panels are assembled, there is the notion of a “reactive design environment”. The user “queries” the system to establish relationships among the design entities. In order to establish these relationships and show traceability, the three panels shall react to specific events performed by the user (i.e. the query), for example the mouse moving, a mouse or keyboard button being pressed, etc.

First-Cut Implementation. The initial solution to this was to use Boolean variables and “if-else” statements to check the value of those Boolean variables. For example, if the mouse was over a specific node in the UML class diagram, then a Boolean called “mouseOverNode” would be set to true. In the engineering model, there would be a method to check to see if the “mouseOverNode” Boolean variable is true. If it is true, then all instances of that class would react to the mouse over event.

While this solution works, it is tedious to implement. As the system size increases and the traceability relationships become progressively complex, there are more items to query; and, as a result, there are more Boolean variables and more “if-else” statements to check the values of the Booleans. Moreover, the use of Boolean variables implies a tight coupling of adjacent systems, which is counter to the requirements shown in Figure 3.1.

Java Delegation Event Model. A more efficient design solution is to use the Java Delegation Event Model (DEM). The DEM is based on the Publish-Subscribe design pattern. Publishers generate and send events, and subscribers register or subscribe to those events from the publishers. When a publisher sends

out or publishes an event, all subscribers interested in that event are notified. The main objectives of Publish-Subscribe are to provide a method of signaling from a publisher to subscribers and to provide a method to dynamically register and deregister subscribers with a publisher. The DEM refers to publishers as “event sources” and subscribers as “event listeners” (Larman, 1999).

In Java, event sources originate or fire events. The event source defines the set of events it creates by providing a set of methods which are used to register specific listeners for those events. Event listeners are objects that implement a specific Java `EventListener` interface. The `EventListener` interface implements one or more methods which are invoked by the event source in response to a specific event (Java, 1997).

The software architecture for the prototype implementation makes exclusive use of two technologies: (1) the JavaBeans framework for establishing graphs of listener-driven events using the DEM; and (2) the Violet UML Editor graphical user interface framework.

3.3 Listener-Driven Events Using the Java DEM

One method of connecting multiple models of visualization together is through the use of event listeners in Java. An event occurs every time a user moves the mouse, presses the mouse button, types a key on the keyboard, etc. The event is represented by an object which identifies the event source and provides information about the event. Any object can be notified of the event; the only

prerequisite is that the object must be registered as an event listener on the appropriate event source. Event sources are typically components or models.

Java Swing components can generate a variety of events. Table 3.1 shows examples of events and their associated event listeners.

Action that Results in an Event	Associated Event Listener
User clicks a button, presses Enter while typing, or chooses a menu item	ActionListener
User closes a frame (main window)	WindowListener
User presses a mouse button while the cursor is over a component	MouseListener
User moves the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Component gets the keyboard focus	KeyListener
Table or list selection changes	ListSelectionListener
Any property in a component changes such as the text on a label	PropertyChangeListener

Table 3.1: Examples of Actions Performed and Their Associated Event Listeners

Any number of event listener objects can listen for any number of events from any number of event source objects. Figure 3.4 shows one listener is registered to be notified of one event from one source. Figure 3.5 shows that there can be more than one listener for a single event from a single source. Figure 3.6 shows that a single listener can listen to events from multiple sources.

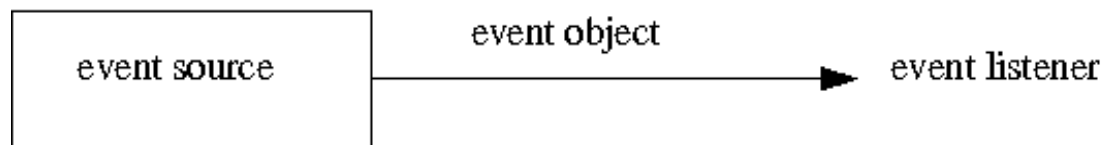


Figure 3.4: One Listener Registered to One Source

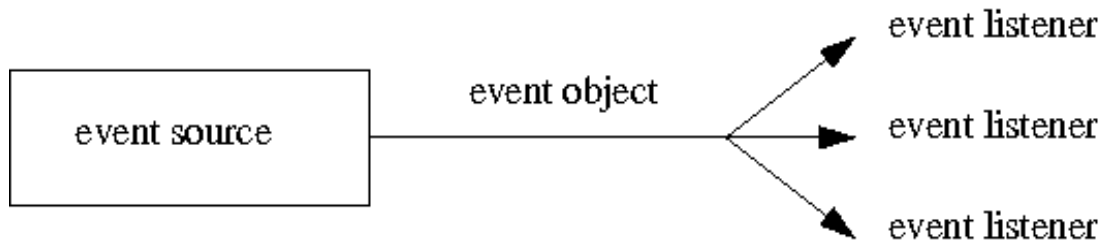


Figure 3.5: Many Listeners Registered to One Source

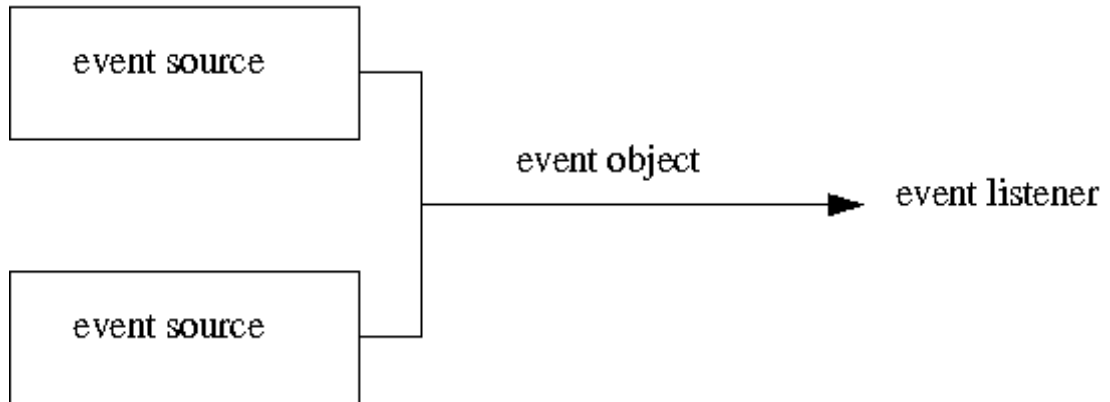


Figure 3.6: One Listener Registered to Many Sources

Every event handler must have the following three pieces of code:

1. In the declaration for the event handler class, the class must either implement a listener interface or extend a class that implements a listener interface. For example:

```
public class MyClass implements PropertyChangeListener {
```

2. An instance of the event handler class must be registered as a listener on one or more components. For example:

```
someComponent.addPropertyChangeListener(instanceOfMyClass)
```

3. The event handler class must implement the methods in the listener interface. For example:

```
public void propertyChange(PropertyChangeEvent e) {
    ...//code that reacts to the property change...
}
```

}

In order to detect any property that changes in a component, a program must have an object that implements the `PropertyChangeListener` interface. The program must register this object as a property change listener on the event source, using the `addPropertyChangeListener` method. When any property in the component changes, the component fires a property change event. This results in the invocation of the property change listener's `PropertyChange` method. The single argument to the method is a `PropertyChangeEvent` object that gives information about the event and its source.

For a more detailed discussion of event listeners, please see Sun's website for "Writing Event Listeners" (Writing, 2006).

Simple Circles Example. Figure 3.7 shows a screendump of a simple example that implements and evaluates groups of event listeners. The "Small Circle" and "Big Circle" rectangles should be thought of as classes that represent concepts appropriate to small and large circles. They would go in the "UML Class Diagram" in Figure 3.2. The right-hand side of Figure 3.7 shows an overly simplified engineering system consisting of ten small circles, ten medium-sized circles, and five large circles.

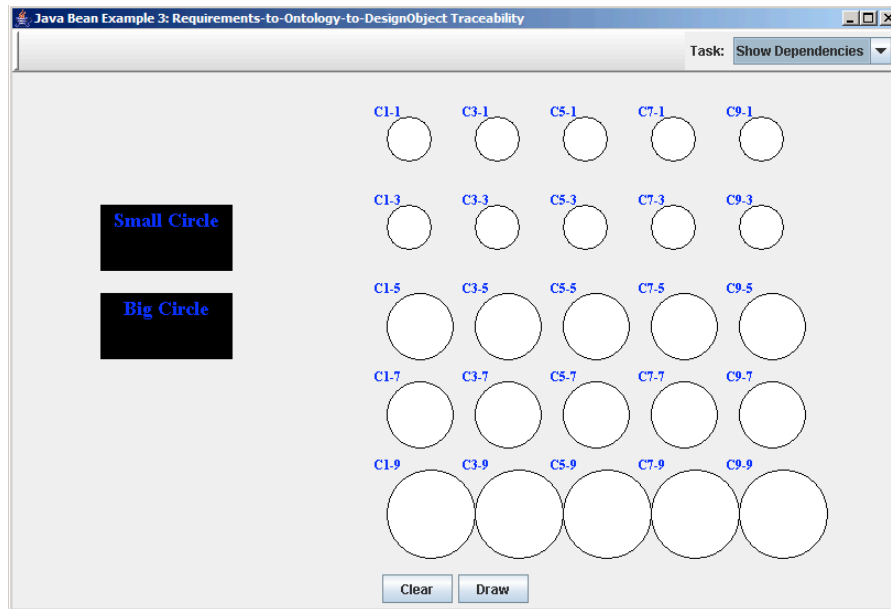


Figure 3.7: Circles Example utilizing Event Listeners

Looking forward, traceability links are established from the “Small Circle” concept rectangle to each of the small circle objects. Similarly, links are established from the “Large Circle” concept rectangle to each of the five large circles. Medium sized circles are considered to have characteristics of both small circles and large circles. Reverse traceability links are created from each object to each of the concepts upon which they are based. In the case of medium circles, links are established from an object back to both the “Small Circle” and “Big Circle” concept boxes.

Setting up the Graph of Event Listeners. The rectangle labeled “Small Circle” registers itself with all the small and medium white circles. The rectangle labeled “Big Circle” registers itself with all the medium and large white circles. All of the medium and large white circles register themselves with the rectangle labeled “Big Circle”, and all of the small and medium circles register themselves with the

rectangle labeled “Small Circle”. Therefore, every circle is listening for an event from the rectangle, and each rectangle is listening for an event from a circle.

In this example, the event source is either the rectangle or the circle and the event is `MouseMoved`. Whenever the mouse moves, an event is triggered; however, a property change event is not invoked until the mouse moves over a rectangle object or circle object. When a change does occur (i.e. the mouse goes from not being over an object to being over an object), a property change event is fired to the object or objects that are listening.

For example, Figure 3.8 shows what happens when the mouse is over the “Small Circle” rectangle. Since the small and medium circles are registered to the “Small Circle” rectangle, they receive notification that a change has occurred on the rectangle. Once the small and medium circles receive the event, they can react to the event. In this case, they highlight green. Likewise, if the mouse is over the “Big Circle” rectangle, the medium and large circles would receive the event, and they would highlight green. This flow of events follows Figure 3.5. There is one event source (the “Small Circle” Rectangle) connected to many event listeners (the small and medium circles).

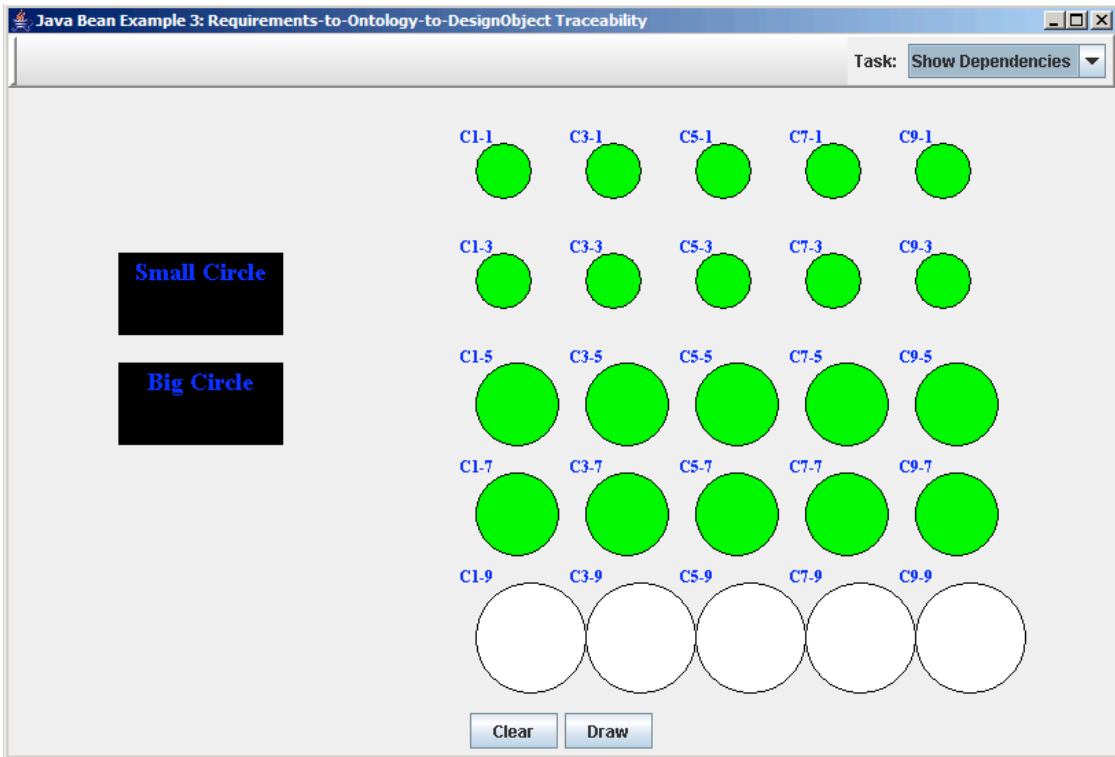


Figure 3.8: System Response for Mouse Over "Small Circle" Rectangle

On the other hand, Figure 3.9 shows what happens when the mouse is over one of the small circles. Since the “Small Circle” rectangle is registered to all of the small and medium circles, it receives notification anytime the mouse is over a small or medium circle. Once the rectangle receives the event, it can react to the event. In this case, it highlights red. Likewise, if the mouse is over a medium circle, both rectangles would highlight red, and if the mouse is over a large circle, the “Big Circle” rectangle would highlight red. This flow of events follows Figure 3.6. There are many event sources (the circles) for one event listener (the rectangle).

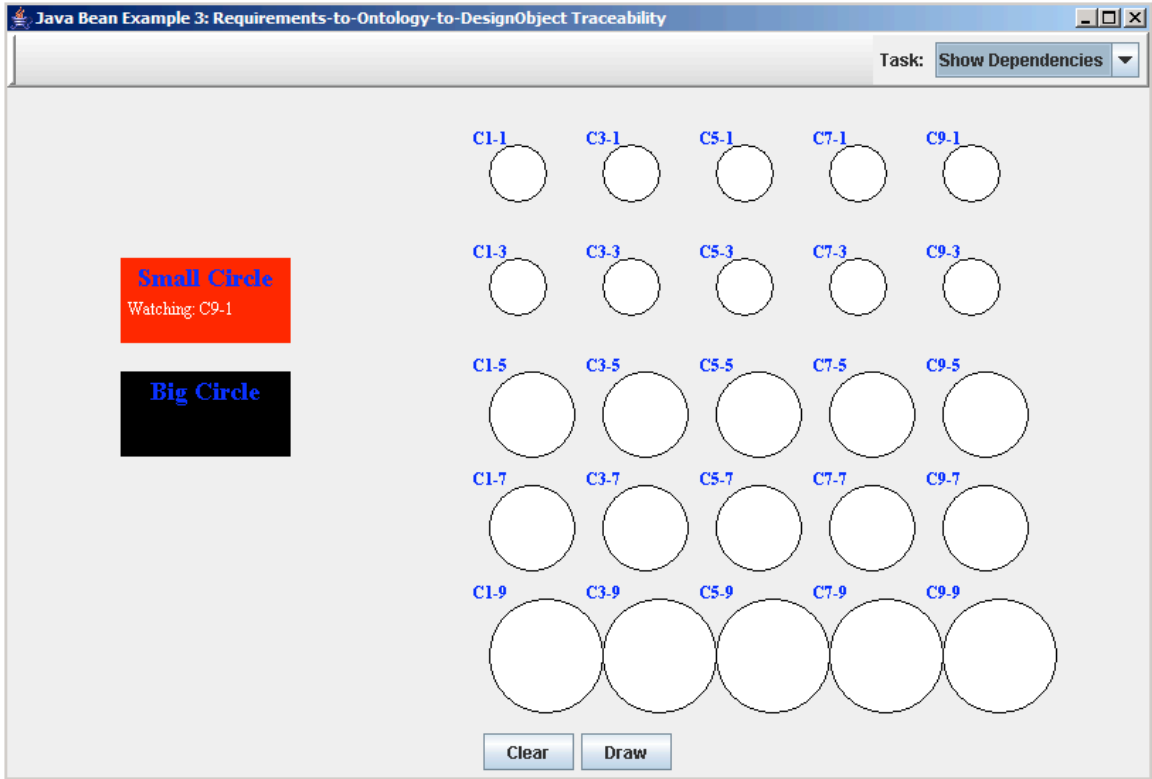


Figure 3.9: System Response for Mouse Over Small Circle

3.4 Violet UML Editor

Violet is a UML editor developed by Cay Horstmann. It supports the drawing of Class Diagrams, Sequence Diagrams, Use Case Diagrams, State Diagrams, and Object Diagrams. It is user-friendly, completely free (including the source code), and is not platform-dependent. Figure 3.10 shows an example of a simple class diagram drawn in Violet. Although Violet has many features, it does not support code generation, reverse engineering, semantic checking of models, and XMI import/export (Violet, 2005).

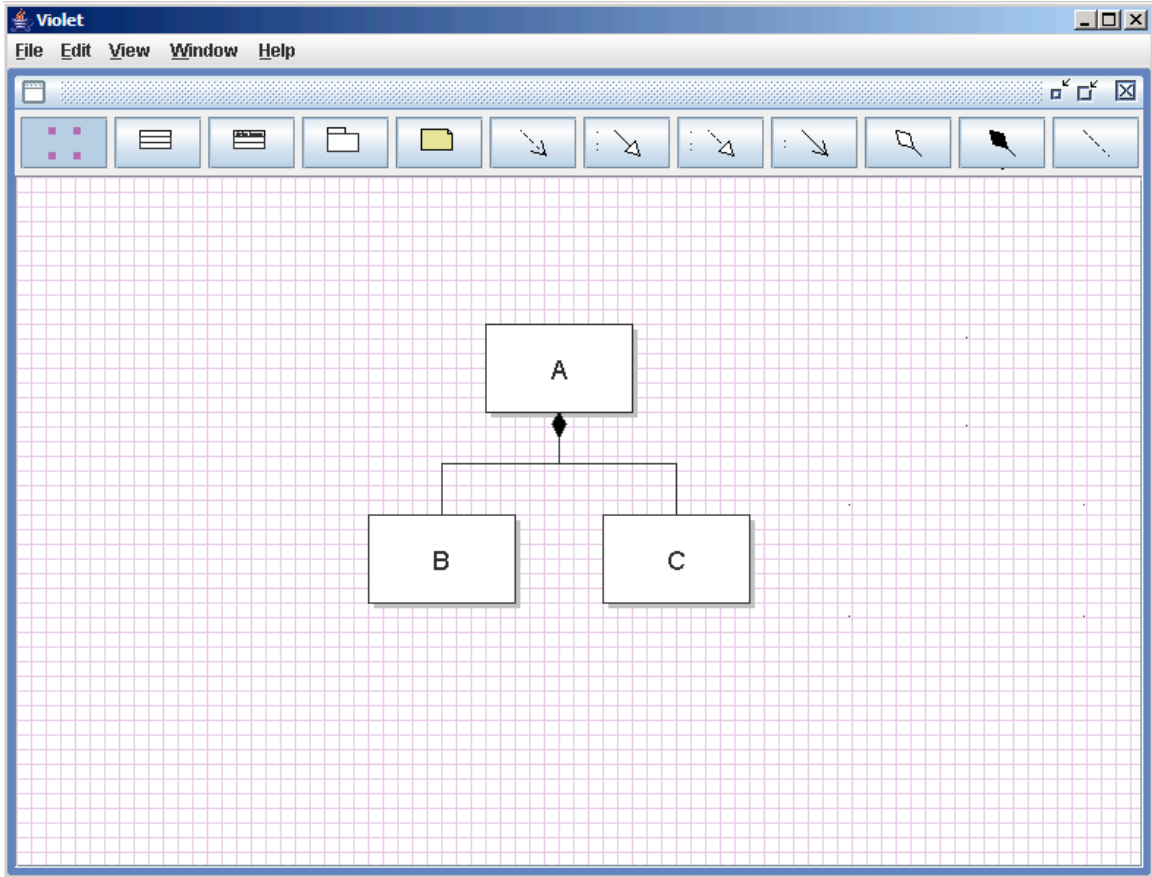


Figure 3.10: Example of a Class Diagram from Violet

Violet is built on top of a graph editor framework. A graph consists of nodes and edges. For example, in a class diagram, the nodes are the rectangles representing a class and the edges are lines or arrows representing relationships between the classes. The graph editor framework encapsulates the features that are common to all graph editor applications. The framework provides a means for specific graph editor applications to express their individual functionality that goes beyond the common features. Violet is a specific application based on this graph editor framework that defines specific behavior for the nodes and edges in various UML diagrams (Horstmann, 2006).

Chapter 4 Application: Washington, D.C. Metro System

4.1 Washington, D.C. Metro System

The Washington, D.C. Metro System is the second largest rail transit system in the United States. The Metro System serves a population of 3.5 million people within a 1,500 square mile area. Figure 4.1 shows the map of the Metro System. The five Metro System lines cover the District of Columbia; the suburban Maryland counties of Montgomery and Prince George's; the Northern Virginia counties of Arlington, Fairfax, and Loudoun; and the Virginia cities of Alexandria, Fairfax, and Falls Church (WMATA, 2006).



Figure 4.1: Washington, D.C. Metro System (Washington, 2006)

The Washington, D.C. Metro System application, as shown in Figure 4.2, illustrates how requirements, a UML class diagram, and an engineering drawing are connected in order to achieve traceability from end-to-end development of engineering systems. The requirements panel is located at the bottom of the window, the UML class diagram is on the left and the engineering drawing is on the right. The engineering drawing is a model of the Red and Green Lines of the Washington, D.C. Metro System.

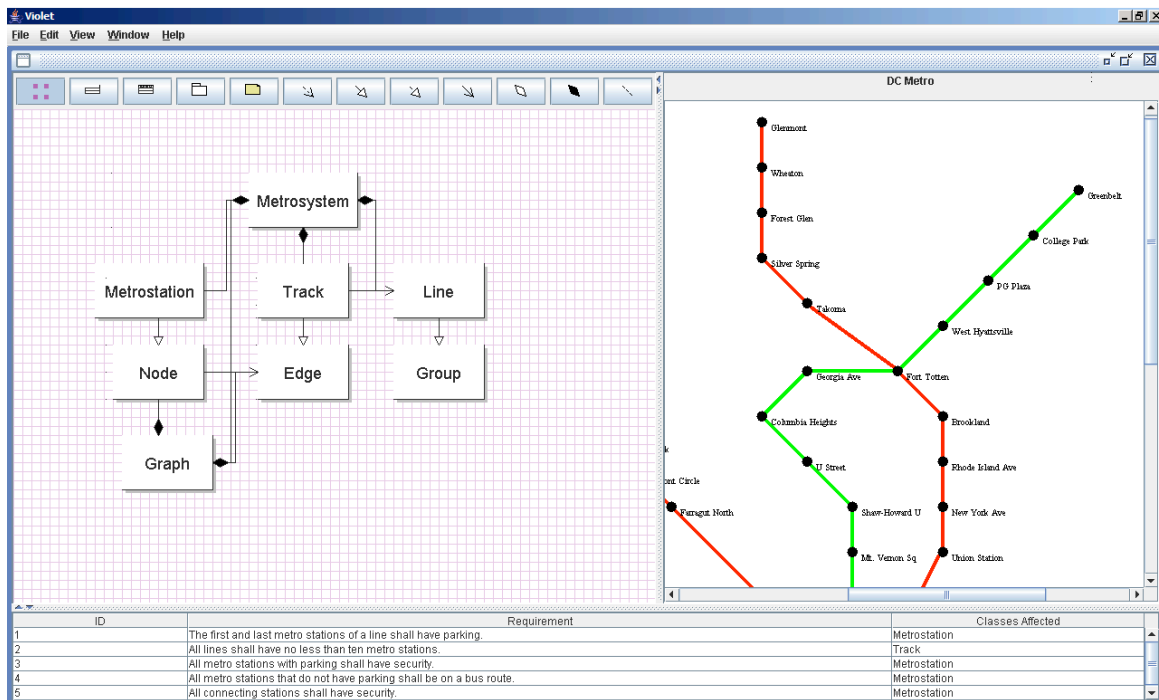


Figure 4.2: Washington, D.C. Metro System Application - Requirements Panel, UML Class Diagram, and Engineering Drawing

4.2 Requirements Panel

The requirements panel consists of a Java JTable, which displays the requirements. The requirements, along with all the classes they affect, are stored in XML format (see Figure 4.3). For the purposes of illustration, the prototype has

only five requirements. As shown in Table 4.1, four of them are satisfied by applying concepts related to the class Metrostation. The remaining requirement is satisfied via concepts in the class Track.

```

<requirements>
  <requirement id="1">
    <text>The first and last metro stations of a line shall have parking.</text>
    <class>Metrostation</class>
  </requirement>
  <requirement id="2">
    <text>All lines shall have no less than ten metro stations.</text>
    <class>Track</class>
  </requirement>
  <requirement id="3">
    <text>All metro stations with parking shall have security.</text>
    <class>Metrostation</class>
  </requirement>
  <requirement id="4">
    <text>All metro stations that do not have parking shall be on a bus route.</text>
    <class>Metrostation</class>
  </requirement>
  <requirement id="5">
    <text>All connecting stations shall have security.</text>
    <class>Metrostation</class>
  </requirement>
</requirements>

```

Figure 4.3: Requirements in XML

ID	Requirement	Classes Affected
1	The first and last metro stations of a line shall have parking.	Metrostation
2	All lines shall have no less than ten metro stations.	Track
3	All metro stations with parking shall have security.	Metrostation
4	All metro stations that do not have parking shall be on a bus route.	Metrostation
5	All connecting stations shall have security.	Metrostation

Table 4.1: Table of Requirements

4.3 UML Class Diagram Panel

The UML class diagram for the Metro System was drawn using Violet (see Section 3.4). As illustrated in Figure 4.4, the Metro System consists of Metro stations, tracks, and lines. Metro station extends node, track extends edge, and

line extends group. Furthermore, a Metro station is a node with additional attributes such as parking, security, etc. A track is an edge with additional attributes such as color. Line is a group with additional attributes such as color (e.g. “Red Line” or “Green Line”). Nodes and edges make up a graph. Nodes are associated with edges and tracks are associated with lines.

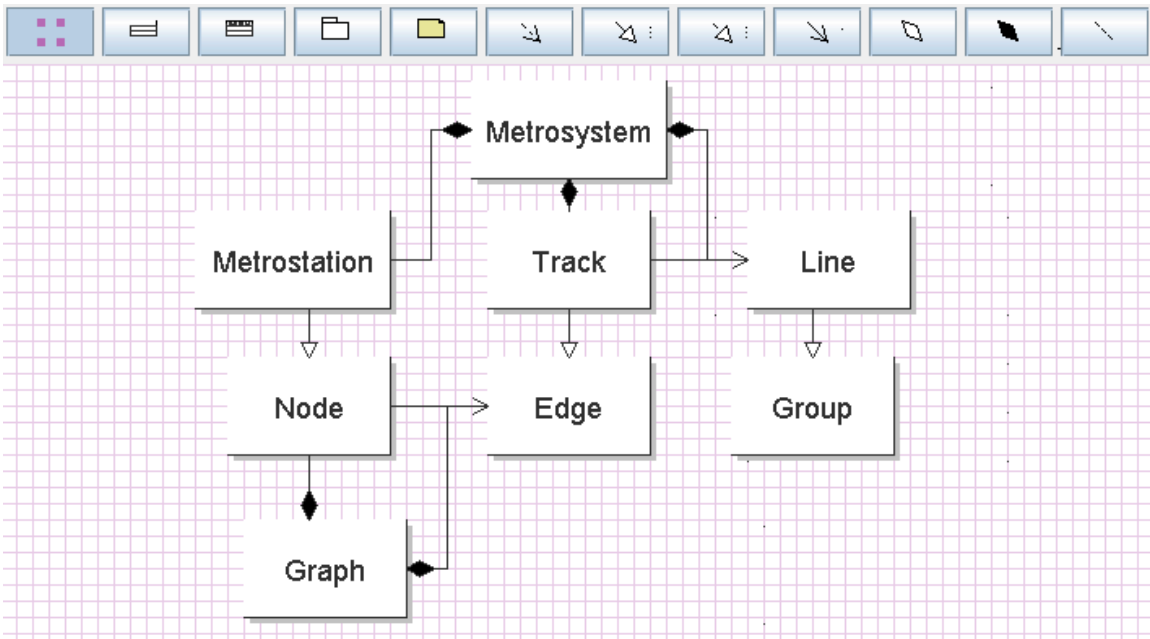


Figure 4.4: UML Class Diagram of Metro System

Violet allows for the importing and exporting of UML diagrams. Therefore, when a UML diagram is drawn in Violet, the file may be saved and re-opened again in Violet. Violet uses XML to export and import the files. Figure 4.5 shows a portion of the XML file after saving the Metro System UML class diagram.

```

<object class="com.horstmann.violet.ClassDiagramGraph">
  <void method="addNode">
    <object id="ClassNode0" class="com.horstmann.violet.ClassNode">
      <void property="name">
        <void property="text">
          <string>Metrosystem</string>
        </void>
      </void>
    </object>
  </void>
</object>
<object class="java.awt.geom.Point2D$Double">
  <void method="setLocation">
    <double>190.0</double>
    <double>0.0</double>
  </void>
</object>
</void>

```

Figure 4.5: UML Class Diagram Saved in XML

Because this example is mostly concerned with class concepts and their relationships, the class attributes and methods are not shown in the diagram. Multiplicity values are not displayed either. However, Violet does allow for the display of attributes and methods, as well as multiplicity values.

4.4 Metro System Engineering Model Panel

Figure 4.6 shows the engineering diagram of the Red and Green lines of the Washington, D.C. Metro System. All information for the Metro System is stored in XML. Figures 4.7, 4.8 and 4.9 show clips of the XML file. The XML file is parsed and the engineering model is drawn according to all the information stored in the XML file.

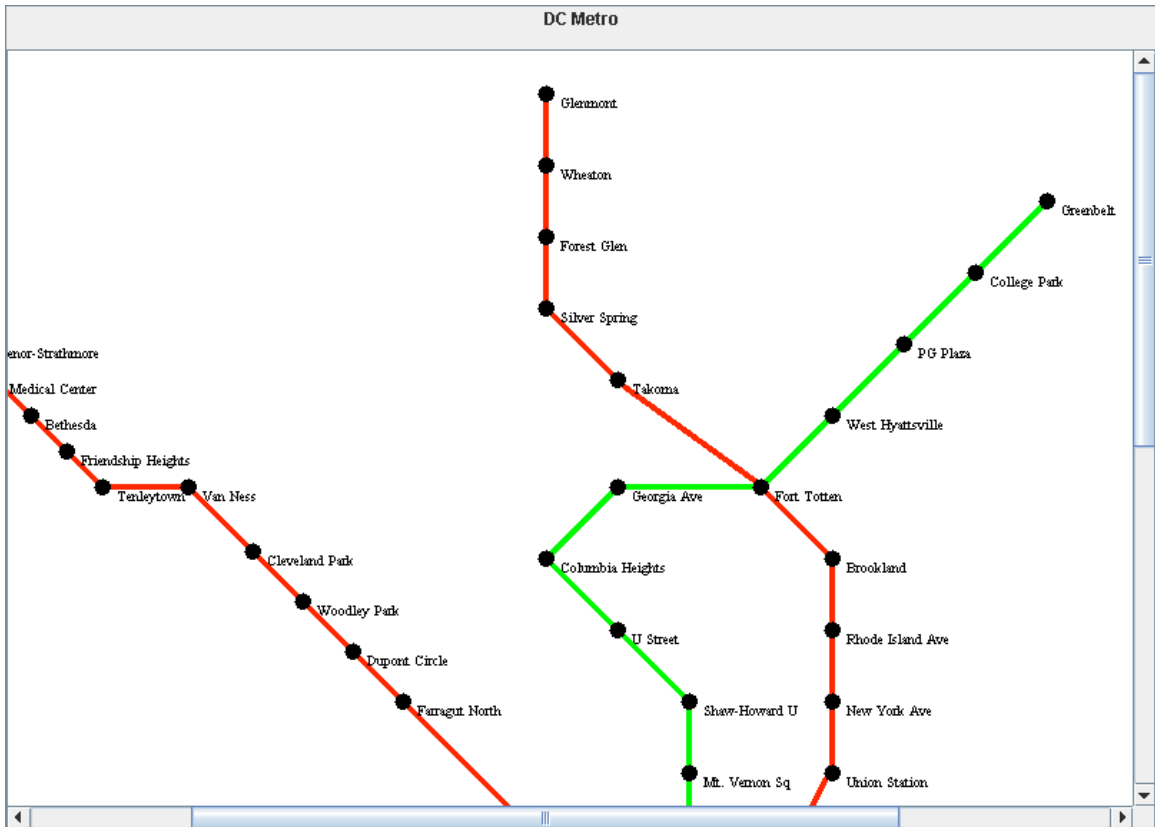


Figure 4.6: Engineering Drawing of the Washington, D.C. Metro System

```

<metrostation id="0">
  <node>
    <name>Greenbelt</name>
    <x>900.0</x>
    <y>100.0</y>
  </node>
  <parking>true</parking>
  <security>true</security>
  <busroute>true</busroute>
  <map>
    <filename>./maps/GREENBLT_MAP.gif</filename>
  </map>
</metrostation>

<metrostation id="1">
  <node>
    <name> College Park </name>
    <x>850.0</x>
    <y>150.0</y>
  </node>
  <parking> true </parking>
  <security>true</security>
  <busroute>true</busroute>
  <map>
    <filename>./maps/COLLEGE_P_MAP.gif</filename>
  </map>
</metrostation>

```

Figure 4.7: XML Fragment for D.C. Metro Stations

```

<track>
  <color>green</color>
  <edge id="0">
    <from> 0 </from>
    <to> 1 </to>
  </edge>
  <edge id="1">
    <from> 1 </from>
    <to> 2 </to>
  </edge>

```

Figure 4.8: XML Fragment for D.C. Metro Tracks


```

<line>
  <color>green</color>
  <group>
    <name>Green Line</name>
    <station>Greenbelt</station>
    <station>College Park</station>
    <station>PG Plaza</station>
    <station>West Hyattsville</station>
    <station>Fort Totten</station>
    <station>Georgia Ave</station>
    <station>Columbia Heights</station>
    <station>U Street</station>
    <station>Shaw-Howard U</station>
    <station>Mt. Vernon Sq</station>
    <station>Gallery Pl</station>
    <station>Archives-Navy Memorial</station>
    <station>L'Enfant Plaza</station>
    <station>Waterfront</station>
    <station>Navy Yard</station>
    <station>Anacostia</station>
    <station>Congress Heights</station>
    <station>Southern Ave</station>
    <station>Naylor Rd</station>
    <station>Suitland</station>
    <station>Branch Ave</station>
  </group>
</line>

```

Figure 4.9: XML Fragment for D.C. Metro Lines

4.5 Traceability Connections

In order to achieve traceability between these visualization models, listener-driven events were used, as described in section 3.3. The requirements are registered with the UML classes which are affected and their appropriate objects in the engineering model that represents the class. For example, if the requirement affects the Metro station class, then the Metro station class node in the UML diagram and the Metro stations in the engineering model are registered to listen to events triggered by that requirement. Likewise, the UML class nodes are registered to listen to events from the objects in the engineering drawing and from

requirements that are affected by that specific class. For example, if the class is track, then it is registered to listen to events from the requirements that affect track and the tracks in the engineering drawing. Also, items that make up the engineering drawing are registered with the class nodes and the requirements that are affected by them. For example, a Metro station node in the engineering model would be registered with the Metro station and node classes in the UML diagram and all requirements that affect Metro stations.

4.5.1 Interaction with the Requirements Panel

When single-clicking on a requirement, the classes that are affected by that requirement are notified of the event. The classes in the UML diagram are highlighted and the items in the engineering drawing are highlighted, because they are registered to listen to the single-click event from the requirement. Figure 4.10 shows a screen shot of tracing a requirement to the UML diagram and to the engineering model. The requirement affects the Metro station, so the Metro station class node in the UML diagram is highlighted and the Metro stations in the engineering model are highlighted.

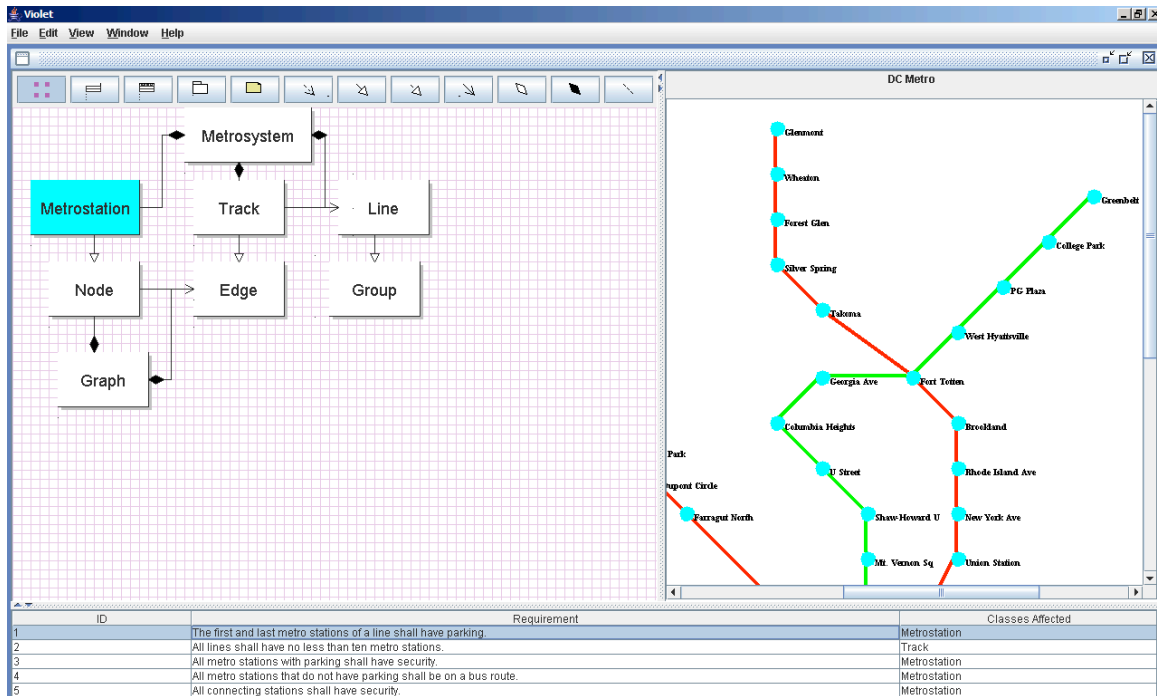


Figure 4.10: Tracing Requirements to UML Class Diagram and Engineering Model

When the requirement is double-clicked, it is verified against the engineering model to make sure there are no errors. The first requirement states, “The first and last Metro stations of a line shall have parking.” Figure 4.11 shows what happens when the requirement is verified and there are no errors. All highlighting still occurs to show traceability, but a window appears that states the requirement has been verified. If there was an error in the engineering model, the pop-up window would state which Metro station or Metro stations violates the parking requirement.

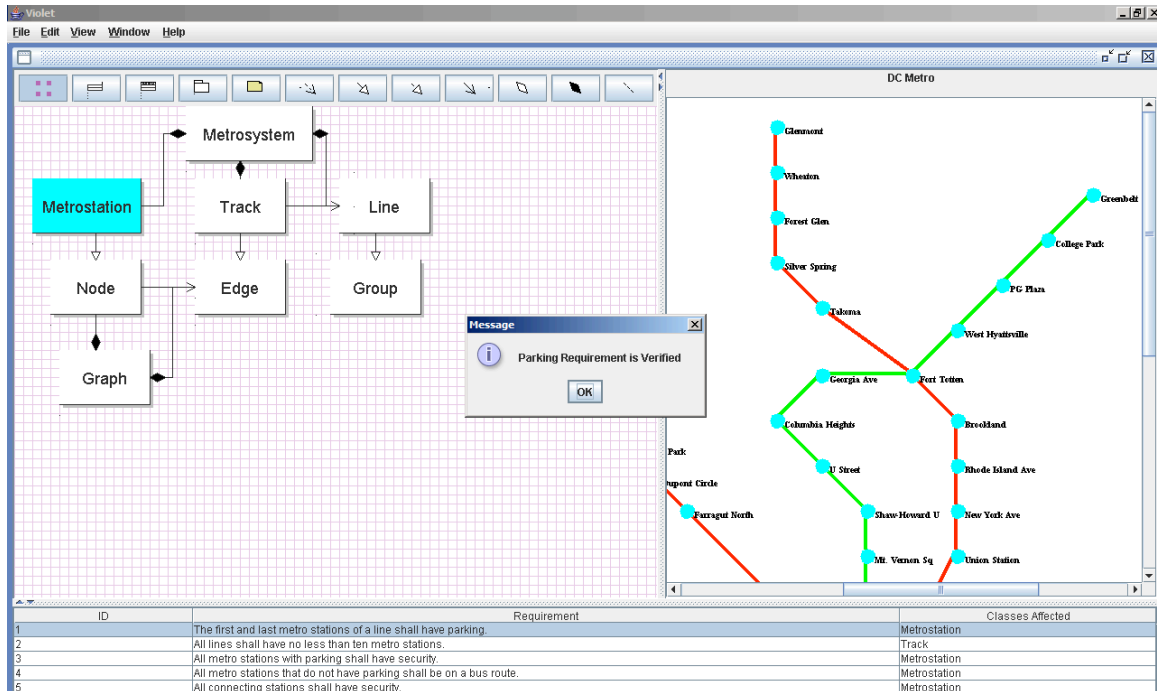


Figure 4.11: Verifying the First Requirement

4.5.2 Interaction with the UML Panel

When mousing-over a UML class node, the engineering drawing objects and requirements that are registered to listen to that event are notified. The objects in the engineering drawing are highlighted and all requirements that affect the class are highlighted because they are registered to listen to the mouse-over event from the class node. Figure 4.12 shows a screen shot of tracing a class node to its objects in the engineering model and the requirements that affect it. When the mouse is over the “Metrostation” class node, all the Metro station nodes in the engineering drawing are highlighted and all requirements that affect the Metro station are highlighted.

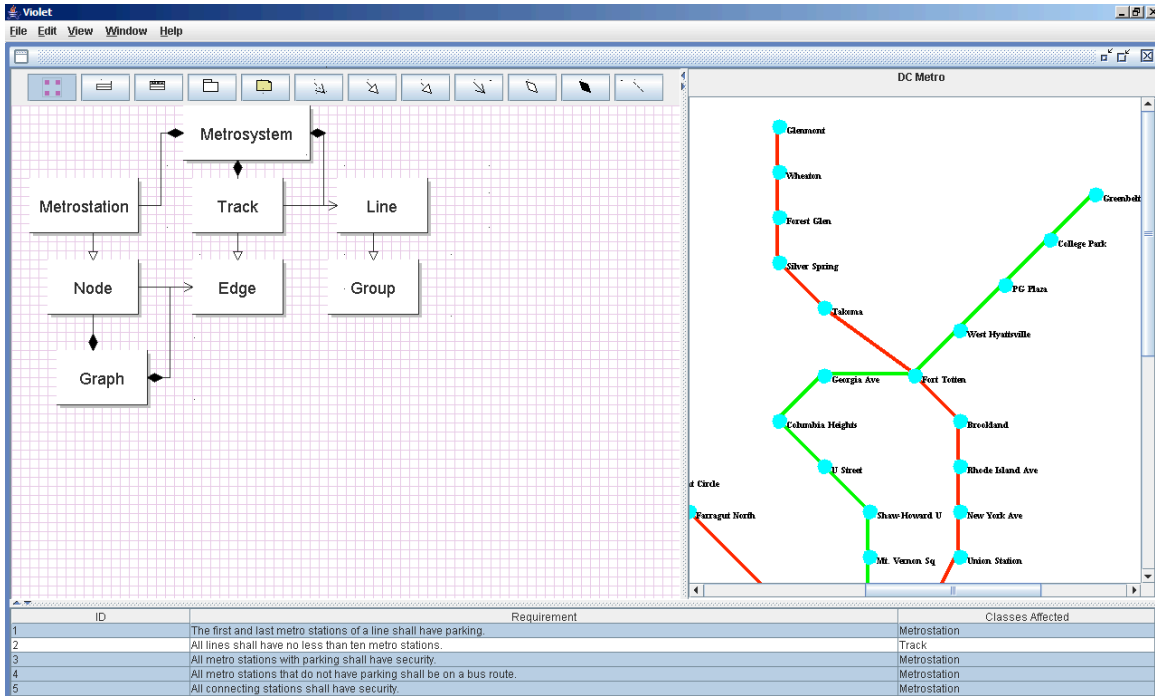


Figure 4.12: Tracing UML Class Node to Objects in Engineering Model and Requirements

4.5.3 Interaction with the Engineering Drawing Panel

When mousing-over an object in the engineering drawing panel, the classes and requirements that are registered to listen to that event are notified. The class nodes in the UML diagram are highlighted and all requirements that affect the class are highlighted because they are registered to listen to the mouse-over event from the class node. Figure 4.13 shows a screen shot of tracing an engineering drawing object to its corresponding class or classes and the requirements that affect that class. When the mouse is over one of the Metro station circles, the “Metrostation” and “Node” classes are highlighted and all requirements that affect the Metro station are highlighted.

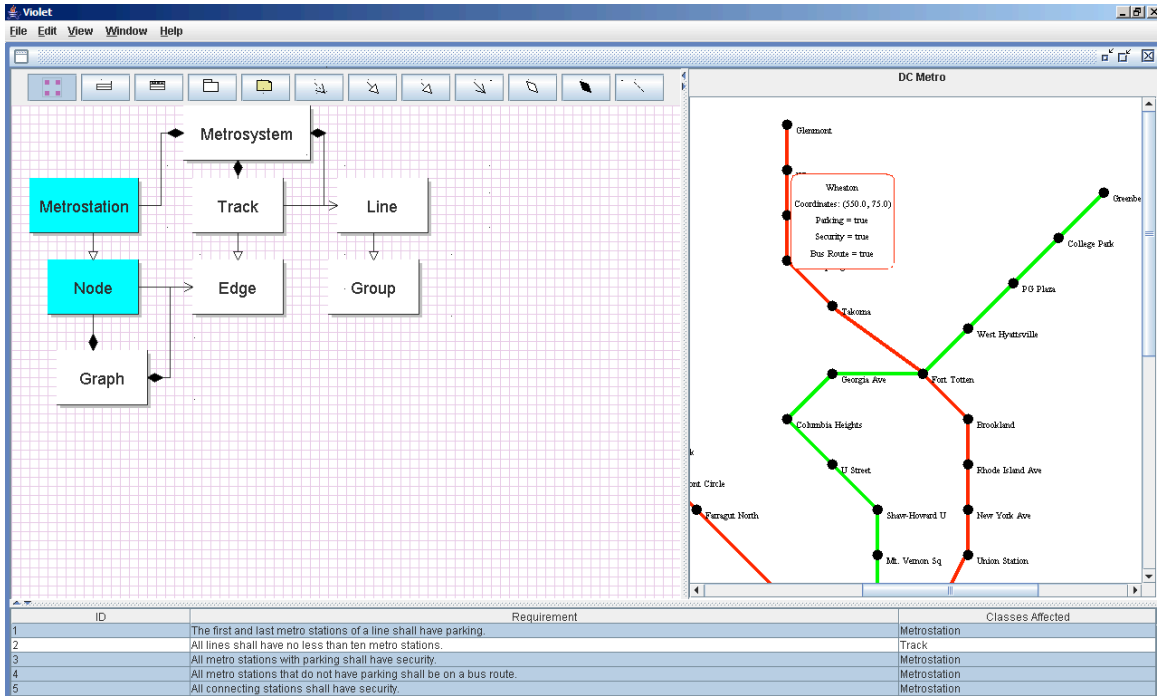


Figure 4.13: Tracing Engineering Model to UML Class Diagram and Requirements

Chapter 5 Conclusion

5.1 Conclusions

The conclusions of this report are as follows:

1. This project has been motivated by the observation and expectation that serious system failures and other software errors can be mitigated with traceability modeling that support validation/verification procedures early in the development lifecycle. Traceability models need to link together multiple models of visualization. The key contribution of this work is preliminary evaluation of a new type of traceability link, where design concepts are inserted between the already connected requirements and engineering objects.
2. Traceability relationships between requirements, design concepts and engineering objects may be arbitrarily complex, possibly forming a very large graph structure. Procedures for establishing these links and responding to external user events need to be efficient and scalable. Here, it has been shown that listener-event models are a suitable approach for creating a variety of traceability relations (e.g., one-to-one, one-to-many, etc.).
3. A key benefit in the “new type of traceability link” is that rule checking procedures may be embedded into design concept nodes. Since individual design concept nodes are part of a larger ontology, rule checking procedures should apply across all projects where the ontology is applicable. Of course, the details of rule evaluation may differ from one technology to the next.

4. This project has exercised these developments through the connectivity of requirements to objects in a model of the Washington, D.C. Metro System. As this study evolved, it became more evident as to how the various system components would communicate. This project has highlighted the importance of XML in connecting components in a multi-model framework.
5. Successful implementation of this proposed approach would require adjustments to the systems engineering processes within companies.

5.2 Future Work

The scope of this project has been restricted to the preliminary implementation of a design problem having a very small number of requirements and a single ontology. Emphasis has been placed on establishing traceability connections. A commercial implementation would require that both of these constraints be relaxed. Thus, future directions of work should include:

1. UML class diagrams are a suitable representation for design ontologies. Key concepts are represented both within the individual classes and the relationships among classes. Only the former has been considered in this study. To account for relationships among classes, work is needed to understand how the constraints implied by relationships (e.g., association, multiplicity, composition, inheritance, etc.) translate to implementation of an executable ontology. Reasoning engines developed to work with Semantic Web languages (e.g., DAML and OWL) might be useful for evaluating axioms with respect to sets of facts and relationships.

2. Austin and co-workers have recently investigated the use of Semantic Web technologies for requirements engineering (Austin, 2006c; Selberg, 2002), developed a framework for using ontologies to support design rule checking (Austin, 2006a), and developed PaladinRM (See Figure 5.1), an interactive tool for working with large graphs of engineering requirements (Austin, 2006b; Mayank, 2004; Mayank, 2005).

In this work, the table implementation of requirements is primitive. Recent work with PaladinRM has shown graphs of requirements can be visualized. Future versions of this work should include replacing the table of requirements with PaladinRM. Appropriate standards for communication are XMI and AP 233 (Oliver, 2002; Muller, 2003; XMI, 2002). Given that future versions of PaladinRM will export/import requirements via AP233, this also implies that the proposed environment will have the capability of working with SLATE or DOORS.

3. This project has assumed that the engineering system description is complete. A natural extension would be to move the environment from one of “supporting analysis” to one of “supporting design.” There is a need to create an interactive drawing environment for the engineering drawing panel so that it can be used for the specification of any type of spatial representation (e.g., transportation systems, building floorplans, network layouts, etc.). The verification of requirements and UML relationships will be built in so that the drawing is checked as the user is drawing the components on the panel.

4. By design, UML provides engineers with visual mechanisms to specify required behavior and structure of software systems. A few additional diagrams are added to UML to create SysML. UML and SysML both do a good job of enabling engineers to specify system components, their connectivity, and pathways of communication that occur in fragments of system functionality. UML and SysML provide little support for dealing with abstract classes and expressing spatial constraints among components. Figures 5.2, 5.3 and 5.4 address the issues with abstract classes and spatial constraints.

The left-hand side of Figure 5.2 shows, for example, a scenario where an abstract class Shape is extended by two concrete classes Rectangle and Circle. A preliminary design representation will be constructed from these simple shapes. The model contains four instances of type Circle. This constitutes a one-to-many relationship. If circle class is moused-over, all of the circle objects should change their appearance. Conversely, if a circle object is moused-over, several behaviors make sense. One possibility is to show a hint attached with the object. How should abstract classes be thought of in this setting? A designer should be able to place an abstract class in the engineering model, knowing that later on, the specific lower-level details will be filled in. How will rule checking work?

With respect to spatial concerns, the left-hand side of Figure 5.3 shows a conceptual model for a simple car – the car will have one body and two wheels. A reasonable design rule would be “check that the wheels are actually

below the car.” At this point it is not immediately clear how spatial queries should be organized.

Finally, Figure 5.4 shows a collection of circular components inside a container. Issues to think about: How to quickly check that the components are actually inside the container?

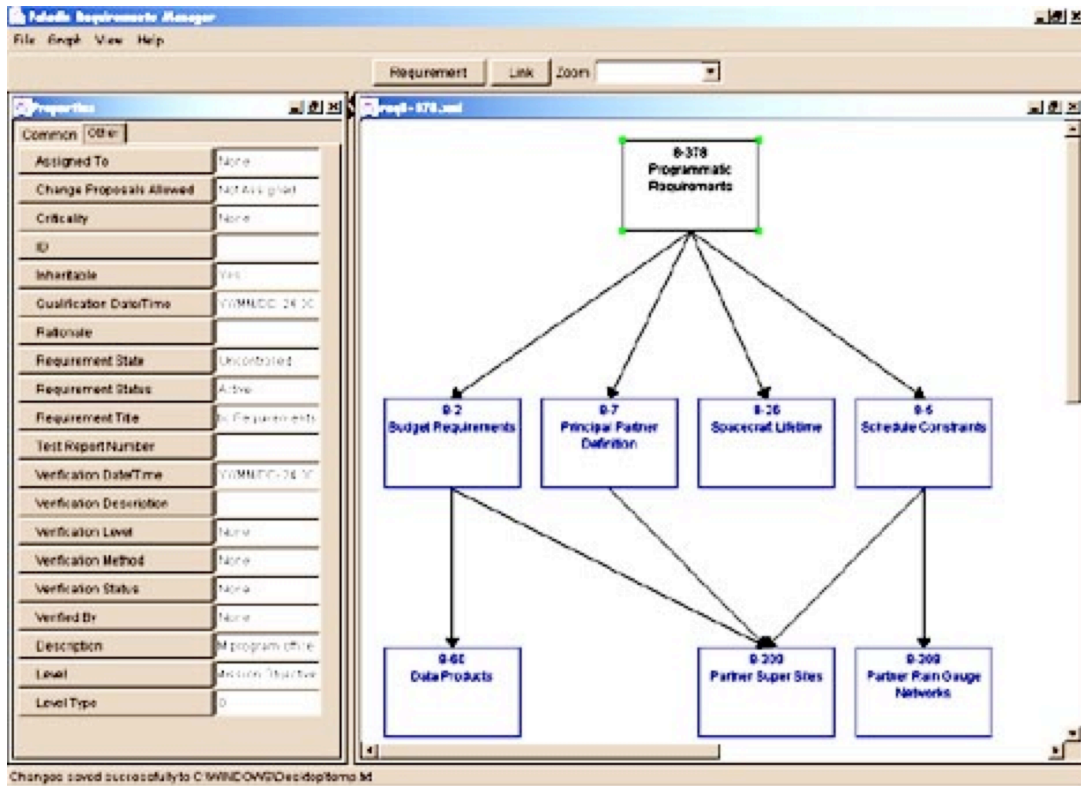


Figure 5.1: Screenshot of the PaladinRM Graphical User Interface (GUI)

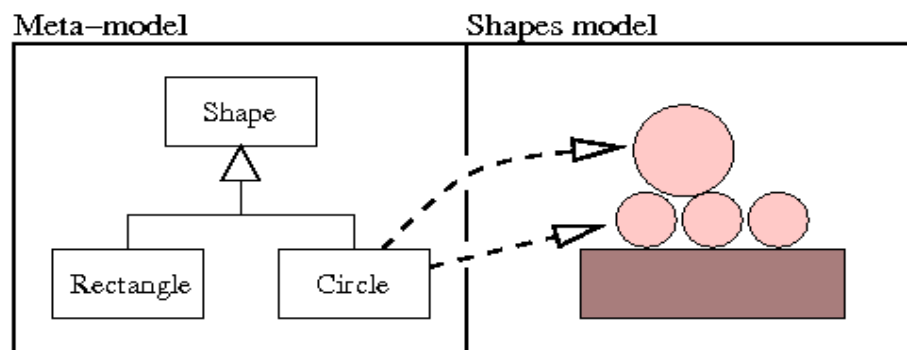


Figure 5.2: Definition and Evaluation of System Assembled from Simple Shapes

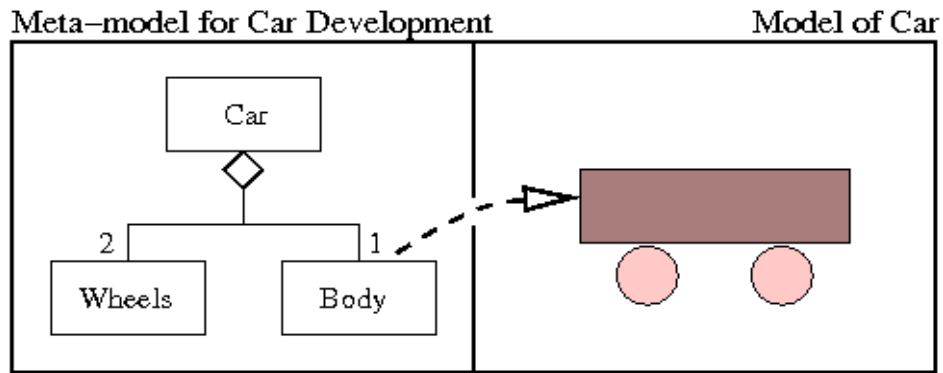


Figure 5.3: Definition and Evaluation of a Simple Assembly

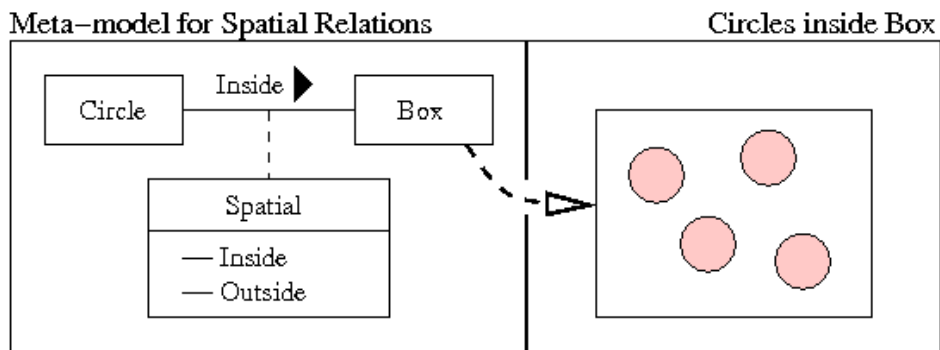


Figure 5.4: Definition and Evaluation of Spatial Relations Among Shapes

In order to prevent catastrophic and expensive engineering failures, this proposal initiates the end-to-end development of engineering system that will occur through multiple models of visualization networked together. Future work will include development of web-centric, graphically driven, computational platforms dedicated to system-level planning, analysis, design and verification of *complex* multidisciplinary engineering systems.

Bibliography

- Austin M.A. Information-Centric Systems Engineering. *Lecture Notes for ENSE 621-622-623*, 2005. Institute for Systems Research, University of Maryland, College Park, MD.
- Austin M.A., Mayank V., and Shmunis N. Ontology-Based Validation of Connectivity Relationships in a Home Theater System. *International Journal of Intelligent Systems*, 21(10):1111–1125, October 2006.
- Austin M.A., Mayank V., and Shmunis N. PaladinRM: Graph-Based Visualization of Requirements Organized for Team-Based Design. *Systems Engineering: The Journal of the International Council on Systems Engineering*, 9(2):129–145, May 2006.
- Austin M.A., Selberg S., and Mayank V. Requirements Engineering and the Semantic Web: Part I Methodology. *Journal of Engineering Design (Submitted for Publication)*, November 2006.
- Baclawski K., Kokar M.K., Kogut P., Hart L., Smith J., Holmes W., Letkowski J., and Aronson M.L. Extending UML to Support Ontology Engineering for the Semantic Web. *UML2001*, 2001.
- Balasubramaniam R., Jarke M.,. Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1), January 2001.
- Barker J. *Beginning Java Objects: From Concepts to Code, Second Edition*. APress, 2005.
- Bentley K., 2003. Does the Building Industry Really Need to Start Over? – A Response from Bentley to AutoDesk’s BIM/Revit Proposal for the Future.
- Berkenkotter K. Using UML 2.0 in Real-Time Development: A Critical Review. 2003. See <http://www.verimag.imag.fr>.
- Berners-Lee, T. XML and the Web. Keynote address at the XML World 2000 Conference.
- Berners-Lee T., Hendler J., Lassa O. The Semantic Web. *Scientific American*, pages 35–43, May 2001.
- Broderbund. 3D Home Architect Design Suite. *Deluxe 6*, 2004. See <http://www.broderbund.com>.
- Cao X., He Z., and Pan Y. Automated Design of House-Floor Layout with Distributed Planning. *Computer-Aided Design*, pages 213–222, 1990.
- Container-Managed Relationships for the 21st Century: EJB 2.0’s new relationship facilities assist developers in rapid deployment, 2002. See <http://www.javaworld.com/javaworld/jw-04-2002/jw-0419-cmp-p2.html>.
- Coupling (computer science). From Wikipedia. See [http://en.wikipedia.org/wiki/Coupling_\(computer_science\)](http://en.wikipedia.org/wiki/Coupling_(computer_science)), 2006.
- Cranefield S. Networked Knowledge Representation and Exchange using UML and RDF. *Journal of Digital Information*, 1(8), February 2001.
- Cranefield S. UML and the Semantic Web. In *Proceedings of the International Semantic Web Working Symposium*, Palo Alto, 2001. See <http://www.semanticweb.org/SWWS/program/full/paper1.pdf>.
- Chun H.W., and Lai M.K. Intelligent Critic System for Architectural Design. *IEEE Transactions on Knowledge and Data Engineering*, 9(4), July/August 1997.

- de Vries B., Jessurun A.J. and vanWijk J.J. Interactive 3D Modeling in the Inception Phase of Architectural Design. *The Eurographics Association*, 2001.
- Downs L. Interchange Format for Symbolic Building Design. Dissertation submitted in partial satisfaction for the MS degree in Computer Science, University of California, Berkeley, CA, 1999.
- Faison T., *Event-Based Programming: Taking Events to the Limit*, Apress, New York, 2006.
- Geroimenko V., and Chen C. (Eds). *Visualizing the Semantic Web: XML-based Internet and Information Visualization*. Springer, 2003.
- Gomez-Perez A., Fernandez-Lopez M., and Corcho O. *Ontological Engineering*. Springer, 2004.
- Guizzardi G., Wagner G., Herre H. On the Foundations of UML as an Ontology Representation Language.
- In Motta E., Shadbolt N., Stutt A., Gibbins N., editor, *Proceedings of 14th International Conference on Engineering Knowledge in the Age of the Semantic Web*, pages 47–62, 2004. (Lecture Notes in Computer Science, 3257) Berlin: Springer - Verlag.
- Hendler J. Agents and the Semantic Web. *IEEE Intelligent Systems*, pages 30–37, March/April 2001.
- Horstmann, Cay. *Object-Oriented Design & Patterns*, pages 334–357, 2006.
- Jackson D. Dependable Software by Design. *Scientific American*, 294(6), June 2006.
- Java AWT: Delegation Event Model. See <http://java.sun.com/j2se/1.3/docs/guide/awt/designspec/events.html>, February 1997.
- Jones, N., 2004. Flawed Drawings caused Spacecraft Crash: Upside-Down Switches stopped Parachutes from Opening, news@nature.com.
- Kharrufa S., Saffo A., and Mahmood W. Developing CAD Techniques for Preliminary Architectural Design. *Computer-Aided Design*, pages 213–222, 1985.
- Kogut P., Cranefield S., Hart I., et al. UML for Ontology Development. *The Knowledge Engineering Review*, 17(1).
- Larman, Craig. Implementing the Java Delegation Event Model and JavaBeans Events in C++. See <http://www.craigarman.com/articles/Delegation%20Event%20Model%20and%20JavaBeans-type%20Events%20in%20C++.htm>, April 1999.
- Liang V.C., and Paredis C.J.J. A Port Ontology for Conceptual Design of Systems. *Transaction of the ASME*, 4, September 2004.
- Mayank V., Austin M.A. Ontology-Enabled Validation of System Architectures. In *Proceedings of Fourteenth Annual International Symposium of The International Council on Systems Engineering (INCOSE)*, Toulouse, France, June 20-24 2004.
- Mayank V., Kositsyna N., and Austin M.A. Graph-Based Visualization of System Requirements Organized for Team-Based Development. Technical Research Report TR 2005-92, Institute for Systems Research, College Park, MD 20742, June 2005.
- Madeen D.A., Palma R.M. Architectural AutoCAD: Drafting, Design and Presentation. 2001.

- Martin J.P. Overview of the EIA 632 Standard – "Processes for Engineering a System (Tutorial G). 2003. EIA 632 Working Group Chairman, Raytheon Systems Company.
- Muller D. Requirements Engineering Knowledge Management based on STEP AP233. 2003.
Oliver D. AP233 - INCOSE Status Report. *INCOSE INSIGHT*, 5(3), October 2002.
- OpenDesign. TurboCAD Deluxe. *Version 10*, 2004. The deluxe edition handles 2d- and 3d drawings, definition of simple solids, and boolean operations. A professional edition is supported by the ACIS solid modeling package.
- Peak R.S., Fulton R.E., Nishigaki I, Okamoto N. Integrating Engineering Design and Analysis using a Multi-Representational Approach. pages 93–114, 1998.
- Rosenman M.A., Gero J.S. Modeling Multiple Views of Design Objects in a Collaborative CAD Environment. *Computer-Aided Design*, 28(3):193–205, 1996.
- Sawyer K. Engineers' Lapse leads to loss of Mars Spacecraft: Lockheed didn't tally Metric Units. *Washington Post*, 1999.
- Selberg S. Requirements Engineering and the Semantic Web. *MS Thesis, Institute for Systems Research, University of Maryland, College Park, MD 20742*, May 2002.
- Sequin C. and Downs L. Symbolic CAD Tools for Architecture. Technical report, University of California, Berkeley, CA, 1997.
- Shanks G., Tansley E., Weber R. Using Ontology to Validate Conceptual Models. *Communications of the ACM*, 46(10):85–89, 2003.
- Staab S., and Maedche A. Ontology Engineering beyond the Modeling of Concepts and Relations. In Benjamins R.V., Gomez-Perez A., Uschold M., editor, *Proceedings of 14th European Conference on Artificial Intelligence: Workshop on Applications of Ontologies and Problem-Solving Methods*, 2000.
- Starr L. *Executable UML (How to Build Class Models)*. Prentice-Hall Publishers, 2002.
- SysML Partners, Systems Modeling Language. Note. This presentation describes the extensions to UML 2 for Systems Engineering. New diagrams types include "requirements" and "parametric" diagrams. November 2003.
- Unified Modeling Language (UML). See <http://www.omg.org/uml>, 2003.
- Violet. See <http://www.horstmann.com/violet>, 2005.
- Washington Metropolitan Area Transit Authority. See <http://www.wmata.com/metrorail/systemmap.cfm>, 2006.
- Web Ontology Language (OWL). See <http://www.w3.org/TR/owl-ref/>, 2003.
- Wieringa R. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, 30(4), December 1998.
- Whitney D.E. Why Mechanical Design cannot be like VLSI Design. *Research in Engineering Design*, 8:125–138, 1996.
- WMATA Facts. See <http://www.wmata.com/about/metrofacts.pdf>, 2006.

Writing Event Listeners. See <http://java.sun.com/docs/books/tutorial/uiswing/events/index.html>, 2006.

XMI. Referenced on April 5, 2002. See <http://www.omg.org/cgi-bin/doc?ad/99-10-02>. 2002.