Organizing and Enabling Domain Engineering to Facilitate Software Maintenance

Christopher W. Pidgeon, Ph.D.
Semantic Designs, Inc.
12636 Research Blvd. C-214
Austin, TX 78759-2200
+1 512-250-1018
cwpidgeon@semdesigns.com

Abstract

Semantic Designs, Inc. is producing a prototype Design Maintenance System (DMS), a fundamentally different software engineering environment. It is a *semantically*-based software engineering environment—one which emphasizes tool-supported design capture and modification. We preserve design knowledge by *requiring* and *enabling* the design to be updated as a natural by-product of maintenance acts. DMS embodies seven key ideas:

1) Software engineering processes must be comprehensive, i.e., include both development *and maintenance*.
2) Software maintenance is *impossible* without domain engineering!
3) The domain definition produced by domain engineering must be *machine processable* in order to achieve any degree of automation.
4) Domain engineering must produce a *set of composable domains* rather than a single, large, monolithic one.
5) Recording how domain knowledge is used in the synthesis of an application constitutes the application's design. A *persistent design* is essential to effective maintenance.
6) Practical domain engineering must be:
   - *incremental*—to scale for maintenance of legacy systems;
   - *cumulative*—should carry into projects other than the one that generated it.
7) Reverse engineering for maintenance should be cast as a domain engineering activity to produce components.

Keywords:

Software Engineering Environment, Domain Engineering, Domain, Design Maintenance, Reuse

Workshop Goals:

We hope to "infect" others with our meme for software maintenance.

Background: DMS, A Domain-based Software Engineering Environment

Semantic Designs, Inc. is producing a prototype Design Maintenance System (DMS), a fundamentally different software engineering environment. It is a *semantically*-based software engineering environment—one which emphasizes tool-supported design capture and modification. We preserve design knowledge by *requiring* and *enabling* the design to be updated as a natural by-product of maintenance acts. Programmers do not touch the code! How is this achieved?

DMS includes a semi-interactive code generation engine. Knowledge for code generation is organized into a network of domains. *Domain Engineering* identifies problem concepts and implementation knowledge comprising the domain interconnection network. A DMS domain expresses:

- problem domain concepts in terms of a domain specific language syntax and semantics; and
- implementation knowledge in terms of transformations—intra-domain optimizations—and refinements—inter-domain concept mappings.

The domain interconnection network identifies *potential* mappings of abstract problem ideas to target implementation ideas. During *Application Engineering*, the software engineer guides choices among alternative mappings. The code generation engine uses domain implementation knowledge to mechanically transform identified abstract domain concepts into target implementations. DMS records the actual transforms used and their justifications as part of the design's history. During a maintenance episode, the software engineer can browse the design and its history to identify and specify changes. Then, DMS is used to install the desired changes by revising the recorded design and its history. This approach yields a continuous, incremental model of software construction—one that explicitly addresses maintenance.

Key Position: Tool-supported Domains are key for the entire software lifecycle—including maintenance.

8) Software engineering processes must be comprehensive, i.e., include both development *and maintenance*.
9) Software maintenance is *impossible* without domain engineering!
10) The domain definition produced by domain engineering must be *machine processable* in order to achieve any degree of automation.
11) Domain engineering must produce a *set of composable domains* rather than a single, large, monolithic one.
12) Recording how domain knowledge is used in the synthesis of an application constitutes the application's design. A *persistent design* is essential to effective maintenance.
13) Practical domain engineering must be:
    - *incremental*—to scale for maintenance of legacy systems;
    - *cumulative*—should carry into projects other than the one that generated it.
14) Reverse engineering for maintenance should be cast as a domain engineering activity to produce components.

Position 1: The goal for DMS is the efficient and effective support for the incremental construction *and* maintenance of large application systems driven by semantics and persistent designs.

DMS is a software engineering environment that records:

What	a system is supposed to do— the *specification.*
How	the implementation does it—the *code.*
Why	the implementation works correctly and meets performance goals—the *design.*

A key feature of DMS is machine interpretable *semantics* (denoting the meaning of programs). This allows DMS to provide specification analyses, assist in the choice of implementations, as well as supply explanations of implementations.

The knowledge required for DMS is organized by problem domain. A *DMS domain definition* includes the following parts:

- Syntax—two equivalent forms for specification:
    - External form—suitable for *human* processing, e.g. string or graphical
    - Internal form—suitable for *machine* processing, e.g. hypergraph
- A means for converting between the two forms for specification syntax:
    - Parser—external form to internal form
    - Unparser—internal form to external form
- Semantics—the meaning of a specification
- Optimizations—how to simplify or elaborate a specification *within* the domain
- Refinements—how to transform a specification *between* domains
- Analyzers—how to measure "interesting" properties of a specification

Position 3: *All* of the elements of a DMS domain are machine processable.

Some Domain Engineering proponents choose to define a domain as a problem area of interest chosen by subject matter, customer application need, community of users, etc. *Practical* problems in a domain must be implementable. This requires experience in solving such problems. Engineers must be able to articulate implementation knowledge. For a DMS domain definition to be considered feasible, we insist on the presence of such knowledge. Otherwise, there is little hope for machine support. The domain is relegated to mere expository roles. It cannot be considered an active element in software synthesis or maintenance. Thus, a DMS Domain realizes two important notions:

1) As a design *product* description it provides a machine processable notation which can be used to express the intentions for a software artifact.
2) As a design *process* description it provides a machine processable means to describe how intentions in one domain notation may be realized in alternative other domain notations.

The DMS environment, then, supports two activities: Domain Engineering and Application Engineering.

DMS Support of Domain *and* Application Engineering

The following signatures characterize the functionality of DMS tools supporting Domain Engineering.

Parser Generator:
$\text{Grammar}_{\text{Domain}} \rightarrow \text{Parser}_{\text{Domain}}$

Unparser Generator:
$\text{Grammar}_{\text{Domain}} \times \text{Rendering Rules}_{\text{Domain}} \rightarrow \text{Unparser}_{\text{Domain}}$

Semantics Definer:
$\text{External Form}_{\text{Domain}} \times \text{Parser}_{\text{Domain}} \times \text{Domain Description}_{\text{OtherDomain}} \rightarrow \text{Semantics}_{\text{Domain}}$

Transform Definer:
$\text{External Form}_{\text{Domain}} \times \text{Parser}_{\text{Domain}} \times \text{Semantics}_{\text{Domain}} \times \text{Domain Description}_{\text{TargetDomain}} \rightarrow \text{Transforms}_{\text{Domain}}$

Performance Analyzer Definer:
$\text{External Form}_{\text{Domain}} \times \text{Parser}_{\text{Domain}} \times \text{Semantics}_{\text{Domain}} \times \text{Domain Description}_{\text{TargetDomain}} \rightarrow \text{Performance Analyzer}_{\text{Domain}}$

Many of these tools aid the domain engineering process by providing consistency checks on proposed Domain Definitions.

The following signatures characterize the tools that were synthesized during Domain Engineering for use in support of Application Engineering.

$\text{Parser}_{\text{Domain}}$:
$\text{External Form}_{\text{Domain}} \rightarrow \text{Internal Form}_{\text{Domain}}$

$\text{Unparser}_{\text{Domain}}$:
$\text{Internal Form}_{\text{Domain}} \rightarrow \text{External Form}_{\text{Domain}}$

$\text{Semantics}_{\text{Domain}}$:
$\{f_{\text{map}}(\text{Internal Form}_{\text{Domain}}, \text{Internal Form}_{\text{OtherDomain}})\}$

$\text{Transforms}_{\text{Domain}}$:
$\text{Internal Form}_{\text{Domain}} \times \text{Location} \rightarrow \text{Internal Form}_{\text{Domain}}$

$\text{Performance Analyzer}_{\text{Domain}}$:
$\text{Internal Form}_{\text{Domain}} \times \text{Internal Form Location} \times \text{Performance Parameters} \rightarrow \text{Performance Value}$

Under the guidance an Application Engineer (who views specifications in their external form) DMS is directed to perform property preserving transformations—optimization or refinement—on internal forms until a final implementation is achieved. The finality here is determined by the Application Engineer who stipulates the properties of a specification which constitute a final implementation, usually called code. Example properties include:

- the specification is expressed in a particular set of target languages (e.g., ANSI C, SQL, POSIX)
- the specification has particular performance values e.g., sort complexity = O(n ln n), interactive-response-time < 5 sec.

Position 5: DMS records the transformational derivation history for the design in order to support subsequent changes. However, unlike naive code regeneration from scratch, DMS uses sophisticated compiler and delta propagation technology to propagate changes through the design and derivation history. This is how we achieve scaleability, but the topic is beyond the scope of this position paper.

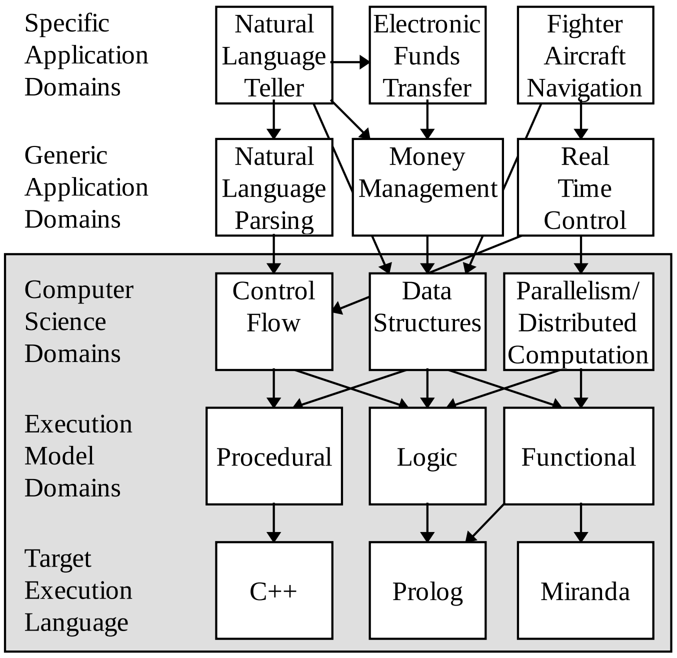
Domain Engineering: in-the-large vs. in-the-small

Position 4: For any particular problem domain, we favor a modest number (~10 to ~100) of modular, specific purpose domains over a small number (~1) of monolithic, general purpose domains. *This allows reuse of most of the very specific domains in new problem areas.*

	Modular	**Monolithic**
Number of Domains	~10 to ~100	~1
Ease of Acquisition	Easier	Difficult
Ease of Application	Easier	Difficult
Reuse Potential	Higher	Lower
Development & Maintenance Cost	Lower	Higher
Repository Management	Tractable	Trivial

In essence, this devolves to the argument between a larger number of more specialized domains having more desirable properties than a smaller number of more generalized domains.

Position 6: We propose a domain interconnection network as exemplified below.



Box = DMS Domain, Arrow = "implemented using"

DMS will be made available with a collection of reusable domains (shaded region) either from the vendor or third parties. Domain Engineers concentrate on defining the value-adding generic and specific application domains that are defined and implemented in terms of the other implementation-oriented domains.

What's a Component?

The prevailing notion is that a component ultimately connotes a fragment of code that realizes some domain concept. Quite often the domain concept is implicit, at least from the perspective of a tool expected to support implementation of the concept. The advantage to this approach is that anyone can write them. The disadvantage is that they can be quite hard to use. Why? There are usually no explicit composition rules. One merely concatenates the code fragments. Another disadvantage is that it is difficult to discern the applicability conditions for the component. Will it work in my configuration? What assumptions is it predicated upon? What constraints does its use impose on subsequent design and implementation decisions?

Position 7: A DMS Component is a triple <*domain*, *concept*, *method*>. The *domain* establishes the problem context—it indicates available background knowledge like other components, performance measures, optimizations, etc. The *concept*, which may be parameterized, is the explicit idea being realized. *Method* is the transformation control plan to implement the concept. It implicitly requires transforms. It may require components at lower level of abstraction. It may be shared with other domain concepts. The composition rules are defined by the transformations, and can be checked when applied.

How Do You Get Components?

Position 7: Reverse engineering is a legitimate, pragmatic domain engineering activity. The process is simple to define, though challenging in practice:

1) Abstract the legacy code through domains.
2) Encode the transformational knowledge to support forward engineering.
3) Generalize and account for component interactions.

By recasting reverse engineering for maintenance as a domain engineering activity, we achieve several desirable outcomes:

- focus is redirected to the production of reusable artifacts (not merely patching the extant code);
- we reduce the cost of future maintenance (because future maintenance activities are able to reuse the efforts of prior ones);
- we should be able to recoup some of the investment in a legacy system (by decrypting and making persistent their domain concepts and construction technologies);
- the stature of maintenance is elevated (by moving from remediation to business value-adding).

Conclusion

Position 2: Domain engineering is essential for comprehensive, anticipatory software production and maintenance.

In sum, a DMS domain captures problem domain concepts *and* construction knowledge. The principal task of domain engineering is the identification of the domain concepts and implementation knowledge, *and encoding that knowledge so it can be reused.* The DMS domain interconnection graph identifies *potential* maps of abstract problem ideas to target execution languages. Since DMS is aware of domains, their interdependencies, and their parts, they can be used by DMS to *mechanically transform* specified domain concepts to code. In the process, the actual transforms and their justifications are recorded as the *design.* Using the *external form in a familiar notation*, both the application engineer and the domain engineer can navigate the design, moving freely across domain notations to *understand how* the code works and *why* it is present in a particular configuration.

Acknowledgements

Construction of a DMS prototype is supported by the National Institute of Standards and Technology, Advanced Technology Program, Component-Based Software Initiative.

References

[1] G. Arango, I. Baxter, C. Pidgeon, P. Freeman, "TMM: Software Maintenance by Transformation", *IEEE Software* 3(3), May 1986, pp. 27-39.

[2] I. Baxter, "Design Maintenance Systems", *CACM* 35(4), Apr. 1992, pp. 73-89.

[3] I. Baxter, "Practical Issues in Building Knowledge-based Code Synthesis Systems", *Proc., Sixth Annual Workshop on Software Reusability*, Owego, New York, Nov. 1993

[4] I. Baxter, "Design (Not Code!) Maintenance", *Proc. ICSE-17 Workshop on Transformational Modification*, Seattle, Washington, Apr. 23, 1995.

Biography

Dr. Pidgeon received a B.S. in Computer Information Systems (1976) and a Masters in Business Administration (1978) from California State Polytechnic University, Pomona. He received a Ph.D. in Computer Science (1990) from the University of California at Irvine where he studied with the Reusable Software Engineering (REUSE) group under the direction of Dr. Peter Freeman (now the Dean of the College of Computing at Georgia Institute of Technology). From 1979 to 1991 Pidgeon taught undergraduate and graduate courses for the Computer Information Systems Department at Cal Poly, Pomona. From 1988 to 1992 Pidgeon was a scientist with Hughes Space & Communications group. From 1992 to 1995 he was a director with Cambridge Technology Partners. Together with Dr. Ira Baxter, he founded Semantic Designs, Inc. in September of 1995. Subsequently Semantic Designs was awarded a three year, $2M grant to develop the prototype Design Maintenance System.