

ModelTalk: When Everything Is a Domain-Specific Language

Atzmon Hen-Tov, Assaf Pinhasi, and Lior Schachter, *Pontis*

David H. Lorenz, *Open University of Israel*

ModelTalk, a model-driven software development framework, supports the creation of product lines by using domain-specific languages pervasively at the core of the development process.

Large-scale, complex, back-end business applications such as telecommunications software constitute a highly competitive and demanding market. These applications feature deep integration with other business and operational support systems. They must be tailored for each customer, and the customized systems must meet strict extrafunctional requirements, commonly called “telco’s five 9s” (99.999 percent availability). All this, combined with the need for an agile development process with a short time to market, presents a software development and business challenge.

One way to cope with this challenge is to construct a software product line.¹ You establish an initial collection of rich, generic, reusable software components. These shared assets of the product line are then methodically assembled, customized, and fine-tuned for each production system. But you must also have a reliable, concise customization process to go with your product line. Such a process is necessary for performing pervasive modifications to the shared assets quickly without compromising their robustness.

Model-driven development (MDD) is often the development process of choice.² This is where domain-specific languages (DSLs) enter the picture.³ DSLs possess two, highly desirable characteristics. First, they facilitate variability management in MDD by providing the means for expressing customization concisely and often declaratively. They also support a producer-consumer development process, in which one group of developers defines a DSL around components and concerns, while other developers use that DSL to de-

claratively implement concrete requirements.

When considering broad adoption of DSLs in the development organization, you should anticipate and carefully plan for increased DSL usage. You wouldn’t want to end up with numerous, disconnected DSLs, requiring your developers to master many concrete syntaxes, development styles, and tools. That might lead to a great cognitive burden, ironically inhibiting the same agility you were trying to achieve by establishing a DSL-based development process in the first place.

Once you transition to DSL successfully and manage DSL scalability wisely, you’ll discover that much of your software development effort has successfully shifted from writing imperative code to writing declarative composition and customization scripts. This is the case with ModelTalk, a DSL authoring and execution framework.⁴

Approach

The ModelTalk motto is “Everything is a DSL.” DSLs can be extended and composed. They’re

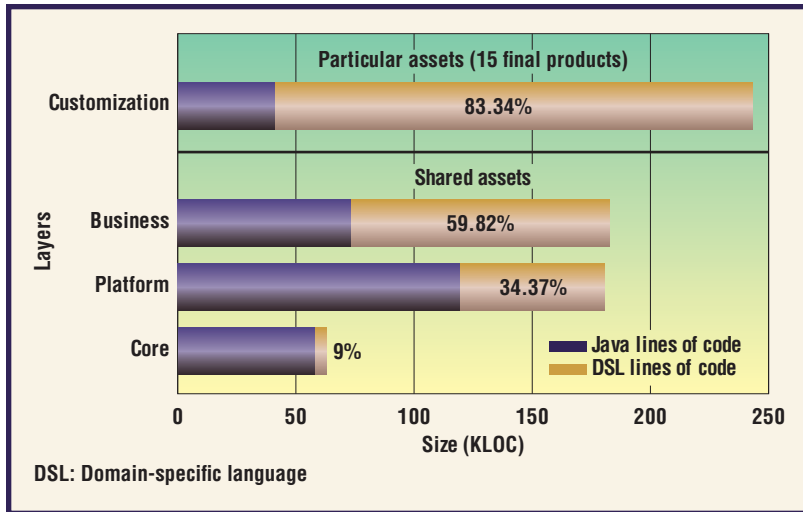


Figure 1. The ratio of Java (purple) to DSL (gold) code by architectural layer in a product line of business support systems. Most of the assembly and the customization of products is done with DSLs.

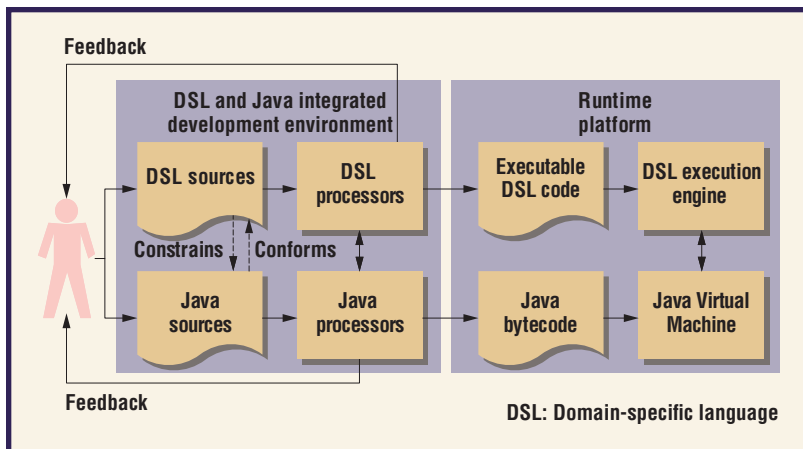


Figure 2. An integrated environment for DSL and Java development. DSL development mirrors Java development, and DSL code governs Java code.

written in (meta) DSLs, specializing and containing instances of other DSLs, all governed by the same syntactical rules.

A system implementation in ModelTalk consists of

- declarative DSL code containing definitions of stateful instances, classes, and metaclasses that describe the system’s structure and configuration; and
- localized Java code containing methods that define the system’s behavior separate from the structure and configuration data.

DSL code is the dominant part of the system. For example, Figure 1 depicts the distribution of DSL and Java code in a subsystem of an actual commercial business support system for the telecommunications market developed with ModelTalk. The ModelTalk architectural layers—Core, Plat-

form, and Business—contain shared assets. At the top layer (Customization), where final products are assembled and customized, 83 percent of the code is written in DSLs and just 17 percent in Java. The relatively large amount of DSL code (approximately 200 Kbytes of LOC) reflects large variation among final products. To achieve similar variability using only Java would have required many more LOC without providing the same compact, precise semantics that DSLs offer.

The ModelTalk approach is based on three design principles that, when combined effectively, enable programming with DSLs to scale up well: textual DSLs, an integrated DSL-Java development environment, and an interpretive approach.

Textual DSLs

In ModelTalk, DSLs are textual rather than graphical. The concrete syntax for DSL code is XML. The ModelTalk core provides a general-purpose language with abstractions (*Class*, *Property*, and so on) and syntax rules for defining DSLs. A developer creates a new DSL by adding new and constraining existing class properties to better describe the domain-specific concepts and terminology. In XML, these are expressed via tag names, which constitute the domain-specific terminology.

Integrated DSL-Java Development Environment

ModelTalk comes with a unified IDE for DSL and Java, implemented over Eclipse (see Figure 2). The DSL development process in ModelTalk is analogous to the Java development process. By providing IDE support for DSL scripting with the same look and feel as for Java programming, ModelTalk achieves instant productivity in the IDE and easier assimilation and adoption of DSL-based development.⁵

The ModelTalk IDE reflects changes in DSL definitions immediately, providing DSL programmers with full support for autocompletion (during editing), navigation, and consistency checking. When developers save their work, the Eclipse IDE automatically invokes the ModelTalk DSL analyzer (a DSL processor in Figure 2), performing incremental cross-system validation similar to background compilation in Java.

After the developers have modeled and captured the domain in a DSL, they implement the behavior in Java. During this stage, the DSL analyzer monitors incompatibilities between the Java and DSL elements and reports violations in the IDE’s standard Problems view. Additional views let developers browse and navigate the DSLs

(between one DSL element and another, as well as between DSL elements and their Java counterparts). A refactoring tool for DSLs is also available, and changes are propagated to Java when necessary.

Interpretive Approach

DSLs in ModelTalk are interpreted; they don't need to be transformed into Java (by a code generator). At runtime, instances of DSL classes are instantiated and used as meta-objects for their corresponding Java instances through a technique called (model-driven) *dependency injection*.⁶ The DSL execution engine is ModelTalk's runtime component, implemented as an inversion-of-control container and a dependency injection framework (in the Spring framework style). It can run either inside a standard J2EE application server (for example, JBoss) or inside a lightweight container.

The execution engine's primary responsibility is to maintain the relationships between the DSL and Java elements. This includes object graph instantiation and a reflection API.⁷ When a client requests a DSL instance, the DSL engine finds the corresponding Java class, instantiates it, and injects the property values into its instance variables. This is applied recursively for injected values of a complex type. The DSL-to-Java mapping allows for DSL classes without a Java counterpart. In such cases, the DSL engine maps the class to its superclass's Java counterpart. Consequently, developers and even users can change DSL definitions at runtime without needing to also change the Java code.⁸

The DSL execution engine eliminates the abstraction gap between the rich ModelTalk meta-model and the one provided in Java, bringing explicit metaclass⁹ and other advanced capabilities into Java.

DSL-Based Software Development

In ModelTalk, the framework developer uses DSLs to declaratively expose variability points in imperative code. This exposure makes the variation management explicit and moves application assembly from the realm of code-level development to that of declarative composition.

DSL Reuse

In ModelTalk, DSLs are interconnected (see Figure 3). A DSL can *specialize* another DSL by narrowing the semantics to a more specific domain. A DSL instance can *contain* an instance of another DSL.

DSL Supply Chain

Traditionally, developers (programmers) produce

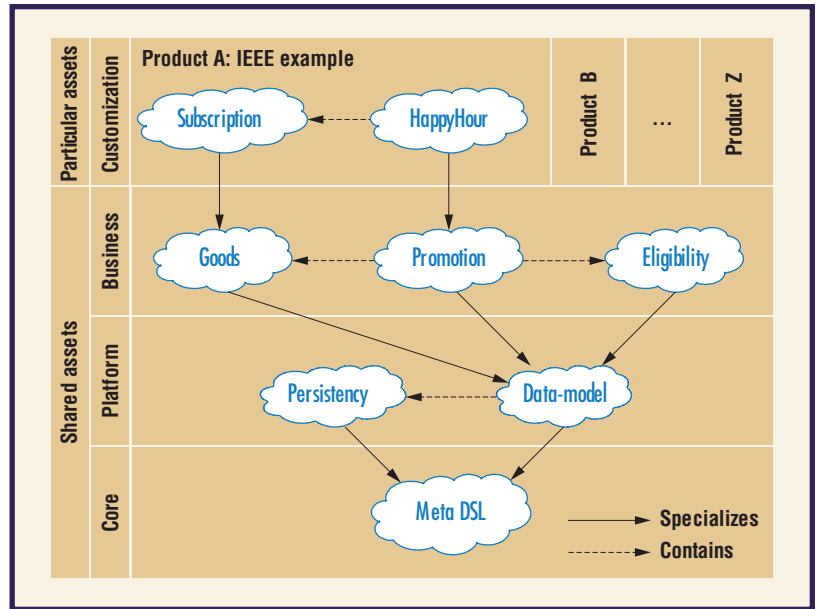


Figure 3. The ModelTalk producer-consumer layered architecture. The bottommost layer defines the most generic Meta DSL. DSLs at higher layers specialize DSLs at lower layers. The topmost layer contains the assembled products (products A through Z).

DSLs for use by domain experts (nonprogrammers). In ModelTalk, however, developers both produce and consume DSLs. Each developer team specializes in a set of domains and shares its expertise with other teams via DSLs. This creates a supply chain of DSLs in the development organization while organizing the DSL artifacts in layers.

All developers are consumers of DSLs residing at the Core layer. The technical experts are producers (and the application and customization developers are consumers) of DSLs at the Platform layer. The application developers are producers (and the customization developers and business experts are the consumers) of DSLs at the Business layer. The customization developers are producers (and the business experts are consumers) of DSLs at the Customization layer.

Architectural Layers

ModelTalk's Core layer resembles the Smalltalk system core. This layer contains the DSL execution engine and provides the basic capabilities for defining and executing DSLs. Because all DSLs in ModelTalk are specializations of a (meta) DSL defined in Core, they all share the same basic syntax and semantic characteristics. This is a key enabler in achieving uniformity in user experience and tooling; for example, autocompletion (during editing), navigation, and consistency checking operate generically on all DSLs. Furthermore, ModelTalk's interpretive nature eliminates the need for tool regeneration when a metamodel definition changes, as is commonly required in other DSL frameworks.¹⁰

In ModelTalk, the framework developer uses DSLs to declaratively expose variability points in imperative code.

The Platform layer defines a rich set of DSL building blocks for creating business applications. Many of the DSLs defined in the Platform layer are bridges to third-party technologies, such as the J2EE standard, the Spring framework, Hibernate, and Apache Axis. Concrete examples of DSLs in this layer include **Persistency** and **Data-model**. **Persistency** wraps the Hibernate framework with a DSL for handling persistency concerns (for example, queries and object/relational mapping). The **Data-model** DSL uses the **Persistency** DSL to define business entities. Wrapping a third-party framework in a DSL hides in effect the technological complexity from the application developers, enabling them to focus on their domain's business logic.

The Business layer is a collection of generic, reusable DSLs and constitutes the business logic. Example DSLs residing here for an online marketing system include **Goods**, **Promotion**, and **Eligibility**. **Goods** defines the products and services offered to the customer. **Eligibility** defines the notion of customer segments and a set of operators that can be applied to them. **Promotion** specializes **Data-model** and uses **Goods** and **Eligibility**. It defines the notion of a product being on sale—that is, the incentive offered, the targeted segment, and the specific goods.

At the top resides the Customization layer, in which the complete customized systems (final products) are assembled mainly from Business-layer DSLs. DSLs from the Platform layer are also used occasionally to customize deep aspects of the system. The Customization layer is managed separately for each customer or group of customers, and customer-specific features are developed here. In the example, **Subscription** and **HappyHour** specialize **Goods** and **Promotion**, respectively, to the IEEE's specific needs.

Variability Management and Evolution

ModelTalk accommodates the evolution of DSLs. A new DSL is initially implemented in the Customization layer as a one-off customer-specific effort. At that stage, the need for variability is limited, so the DSL exposes only a few variability points. Gradually, as new requirements accumulate, the DSL developer identifies and implements new variability points. When the DSL developer recognizes a potential for significant reuse, additional development effort is put into pushing the DSL upstream in the supply chain—that is, moving the DSL to a lower layer. The DSL evolves from being a particular asset to being a shared asset.

A variability point in ModelTalk is essentially a DSL class property. When DSLs evolve, the IDE

alerts the developer about inconsistency between existing DSL instances and their class definition.

Implementing DSLs in ModelTalk

To explain the steps in defining instances, properties, classes, methods, and metaclasses in ModelTalk, we'll use a simplified example of an online marketing system that business experts use to launch targeted promotions on specific goods.

Instances

Figure 4a shows a DSL instance named **IEEE_HH**. This instance “is kind of” **Promotion** (in the Smalltalk sense)—specifically, a member of **HappyHour** (see the listing in Figure 4b). It defines a happy-hour discount on subscriptions to *IEEE Software* targeted at students: \$20 off during October 2009. While specifying this promotion, the DSL programmer is constrained by the type **HappyHour**. All properties must be assigned a value—that is, who is eligible for the promotion (Figure 4a, lines 2–6), the campaign time frame (lines 7–10), the incentive given to buyers (lines 11–14), and the goods participating in the promotion (line 15).

Properties

When editing a DSL instance, you may assign values only to properties that are defined in the DSL class. The assigned value must be of a type that matches the constraints defined in the DSL class. For properties of a complex (user-defined) type, you must specify an explicit **type** attribute. In the **reward** property (Figure 4a, line 11), any subclass of **BenefitGiver** can serve as the type. For example, the menu in Figure 5e lists **FixedAmountDiscount**, **FixedPercentageDiscount**, and **SubscriptionPeriodExtension** as subclasses of **BenefitGiver**.

Choosing the type of a complex property is significant. At the semantic level, the type determines the system behavior. At the mechanical level, the type determines the inner properties that the user should fill in. Setting **FixedPercentageDiscount** as the type of benefit instead of **FixedAmountDiscount** (Figure 4a, line 11) would require the user to specify the discount percentage instead of the amount and currency. The ModelTalk IDE autocompletion feature relieves the programmer from memorizing the valid choices. This composition process is recursive, supporting the definition of nested instance graphs. To promote reuse, ModelTalk also supports references to external instances as an alternative to inline definitions; see, for example, the reference to **IEEE_Software_Subscription** (denoted by the suffix **Ref**—that is, by **business:SubscriptionRef** in Figure 4a, line 15).

Figure 4. Three example DSL instances: (a) a DSL instance named `IEEE_HH`; (b) a DSL class named `HappyHour`; (c) a DSL metaclass named `EntityClass`. The ModelTalk metalevel architecture treats instances, classes, and metaclasses uniformly.

Classes

The syntax for a DSL class definition is the same as for instances. For example, `HappyHour` (see Figure 4b) is an `EntityClass` (and a “kind of” ModelTalk class). It **extends** the `Promotion` class, thus inheriting its structure and behavior. It defines an additional `dateOfPurchase` property (Figure 4b, lines 10–16) in which users can set the effective time of the promotion campaign.

Methods

The DSL producer writes the `HappyHour` behavior directly in Java (see Figure 6a). The ModelTalk DSL engine combines on demand the structural and the behavioral definitions. For example, when a client code requests a DSL instance, perhaps by invoking `ModelTalk.getInstance("IEEE_HH")`, the `HappyHour` class in Figure 6a is instantiated in the Java Virtual Machine and injected (via dependency injection) with the values specified in `IEEE_HH` (see Figure 4a). The injected values are then readily available to the client. For example, the call `getDateOfPurchase` (see Figure 6a, line 5) returns a `TimeWindow` instance (see Figure 6b), injected with the `dateOfPurchase` value (Figure 4a, lines 7–10). This object is then used to determine whether the end user is eligible for the promotion.

The `HappyHour` promotion also exemplifies how declarative composition controls the system behavior. The specified `dateOfPurchase` constraint could be an instance of any subclass of `ScheduleDef` (Figure 4b, line 12), including `TimeWindow`, `Weekends`, and so on. The choice is specified by the type attribute in `IEEE_HH` (Figure 4a, line 7). The DSL user may configure different `HappyHour` instances to have different types of `dateOfPurchase`. For example, the user can declare `dateOfPurchase` to be of type `Weekends`, in which case the `isInTimeFrame` method of `Weekends` is invoked.

Metaclasses

The `HappyHour` listing in Figure 4b is an instance of a specialized metaclass for persistent classes, called `EntityClass` (see Figure 4c). `HappyHour` specifies the persistence trait using the `Persistency` DSL (Figure 4b, lines 2–8). The same rules that apply to instance composition also apply to classes (and metaclasses).

```

01 <Promotion ID="IEEE_HH" type="ieee:HappyHour">
02   <eligibility type="business:EligibilityBySegments">
03     <includedSegmentsList>
04       <item type="business:SegmentRef" ref="IEEE_Students"/>
05     </includedSegmentsList>
06   </eligibility>
07   <dateOfPurchase type="business:TimeWindow">
08     <fromTime>2009-10-01T00:00:00</fromTime>
09     <toTime>2009-11-01T00:00:00</toTime>
10   </dateOfPurchase >
11   <reward type="business:FixedAmountDiscount">
12     <amount>20</amount>
13     <currency>USD</currency>
14   </reward>
15   <product type="business:SubscriptionRef" ref="IEEE_Software_Subscription"/>
16 </Promotion>
(a)

01 <Class ID="HappyHour" type="platform:EntityClass" extends="business:Promotion">
02   <persistenceSpec type="platform:ClassPersistencySettings">
03     <tableName>HAPPYHOUR</tableName>
04     <fetchStrategy type="EagerFetcher">
05       <fullDepth>true</fullDepth>
06       <retrieveReferences>>false</retrieveReferences>
07     </fetchStrategy>
08   </persistenceSpec>
09   <properties>
10     <property type="core:ComplexType">
11       <name>dateOfPurchase</name>
12       <type>ScheduleDef</type>
13       <description>
14         Schedule in which the happy hour is active
15       </description>
16     </property>
17   </properties>
18 </Class>
(b)

01 <Class ID="EntityClass" type="core:Class" extends="core:Class">
02   <properties>
03     <property type="core:ComplexType">
04       <name>persistenceSpec</name>
05       <type>PersistenceSettings</type>
06     </property>
07   </properties>
08 </ Class >
(c)

```

Evaluation and Discussion

Since ModelTalk’s inception, we’ve been collecting data (duration of edit-execute cycles, compilation times, error logs, and other code metrics) to assess its effectiveness. Figure 7 presents the growth over time of the code base (net user-written code,

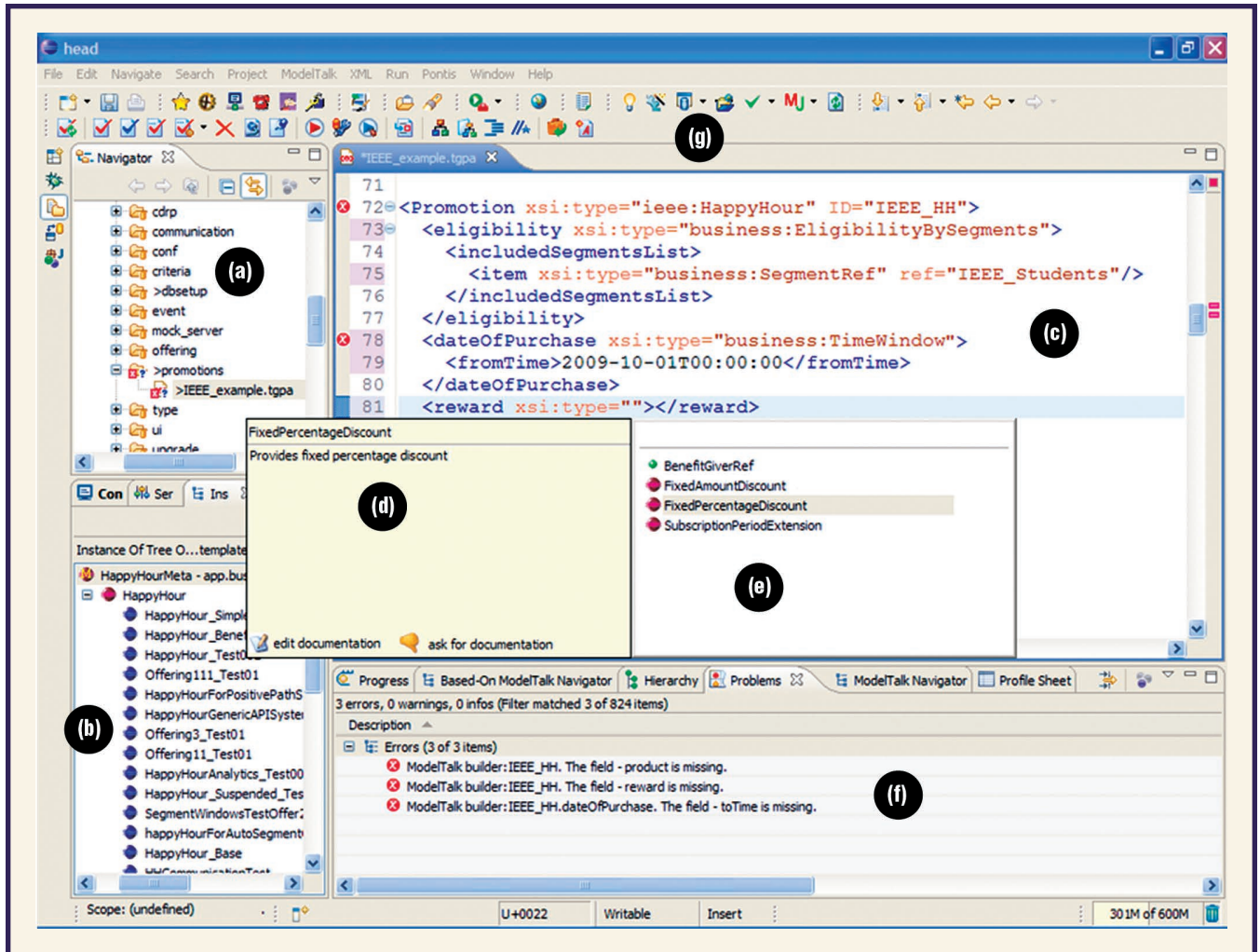


Figure 5. The ModelTalk development environment. A set of Eclipse plug-ins offers Java-like programming tool support for DSL authoring: (a) Eclipse standard navigator view; (b) ModelTalk instance-of hierarchy view; (c) ModelTalk source code editor; (d) ModelTalk documentation tool tip; (e) ModelTalk autocompletion menu; (f) Eclipse problems view; (g) ModelTalk toolbar.

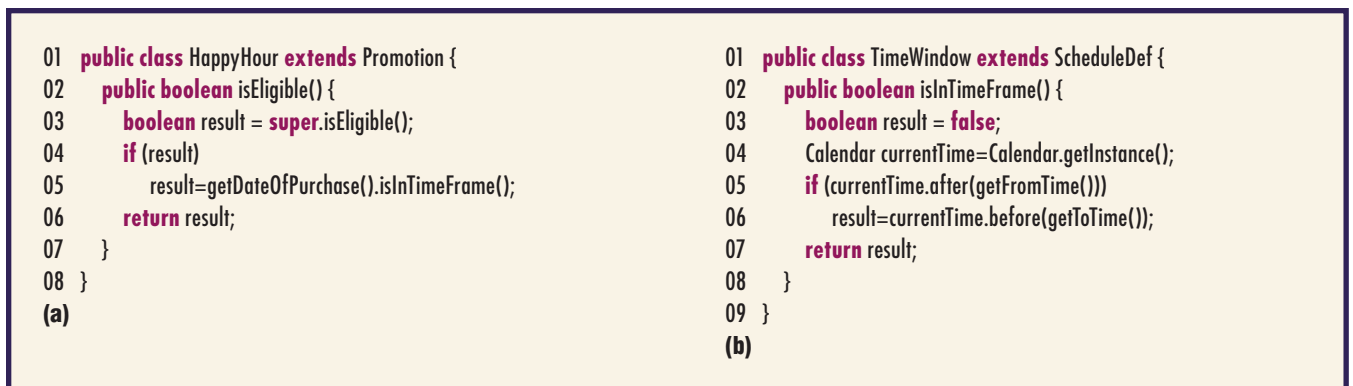


Figure 6. Java code samples show the behavior of (a) HappyHour and (b) TimeWindow instances.

excluding tool-generated code) per architectural layer. System evolution, new features, and enhancements account for the growth in shared assets (in the Core, Platform, and Business layers). New customers and customized products account for the ag-

gregated code growth in the Customization layer.

DSL code has both the largest share in our code base as well as the sharpest growth rate. This indicates that DSLs are at the center of gravity in the ModelTalk-driven development process. This

observation is reinforced by the near absence of unmanaged code (Java code not governed by ModelTalk) in the Business layer.

Currently, the framework contains 6,327 classes, of which 292 are metaclasses. Tens of thousands of instances are expressed in 381K lines of DSL code. Users have done most of the customization (83 percent of Customization LOC) declaratively, further indicating that the ModelTalk approach scales well.

Organization Perspective

An organization considering adopting an approach similar to ModelTalk should take into account a substantial initial investment in building the infrastructure and tools. In ModelTalk, the initial investment was more than 10 person-years. There's also an ongoing cost for maintaining the development environment.

From the human-resources and skills-set perspective, metaprogramming tends to be highly abstract and generic, so it requires highly capable individuals who might also need an adjustment period before becoming productive. We also learned over time that our approach is less effective in domains for which rich graphical tools are mature, such as user interface design and online analytical processing.

Nevertheless, once a development approach such as ModelTalk is in place, the benefits for the organization are tangible. Specifically, the time-to-market and the cost of producing individual, customized products drop significantly. In a particular project, we needed to integrate a system with eight other systems and support up to 200 transactions per second per machine. We went from kickoff to a live system in 17 calendar weeks (10 person-weeks of customization work, see Figure 8).

Developer Perspective

Overall, developers report satisfaction using ModelTalk and specifically praise the short edit-compile cycle. Our data corroborate this: the average incremental build time is less than 10 seconds per build. Customization developers enjoy especially short build times because changes in the top layer have less impact on the system. To maintain these short build times, we periodically devote development resources to improving the DSL processors. Developers appreciate the fact that DSL scripting and Java programming take place in a unified, integrated environment and that they can work on incomplete or inconsistent models. Developers express some dissatisfaction with the lack of diagramming capabilities or interoperability with UML modeling tools.

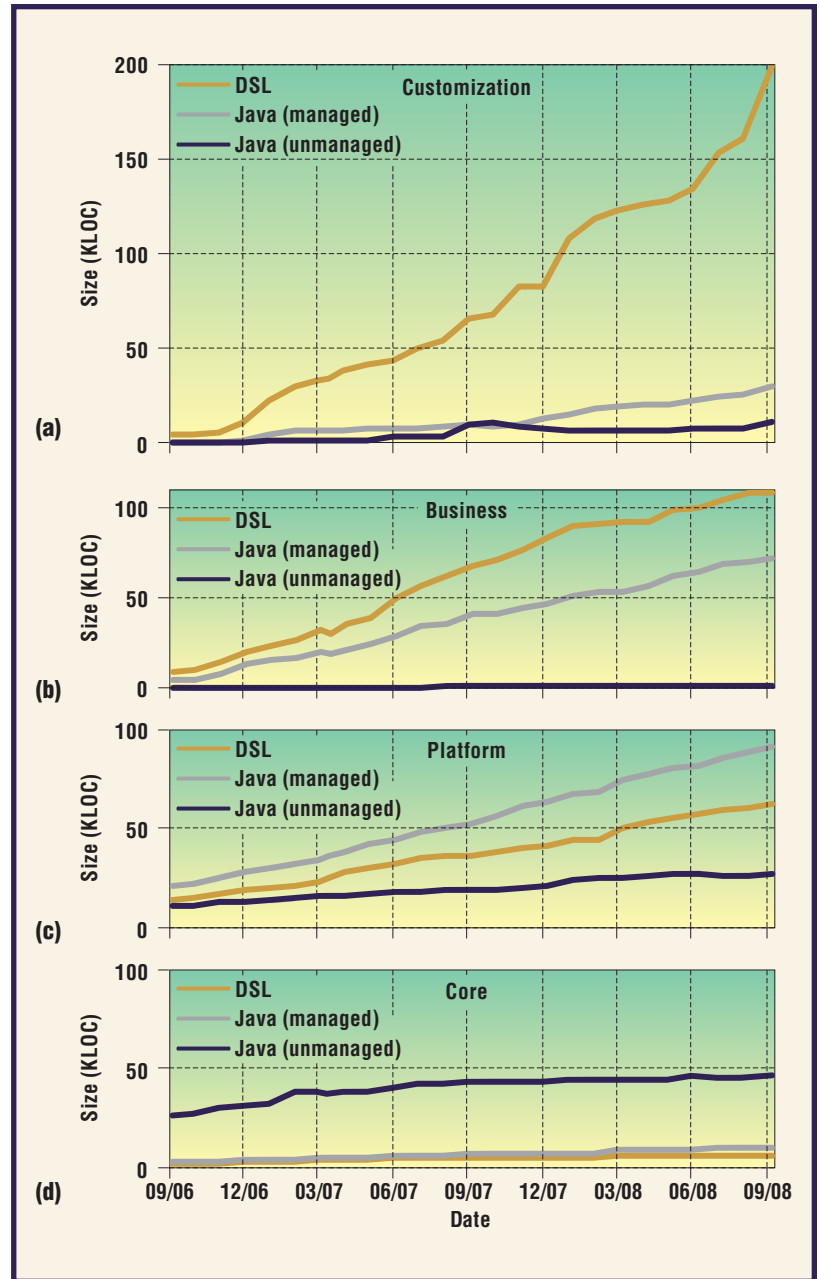



Figure 7. The amount of different kinds of code in each layer: (a) Customization, (b) Business, (c) Platform, (d) Core. The gold lines represent DSL code, the gray lines represent managed Java (Java code governed by ModelTalk), and the blue lines represent unmanaged Java code.

Domain-specific languages aim to improve the development process by raising the level of abstraction around a specific area of the problem. Typically, developers introduce and apply DSLs on a per-case basis only after identifying a specific component or concern that might benefit from having a DSL around it. In large systems, this approach might result in a number of ad hoc, disconnected DSLs. Our work here shows the benefit in embracing DSLs not only for a small number of components or concerns but also as building blocks for systematically constructing large software systems. 

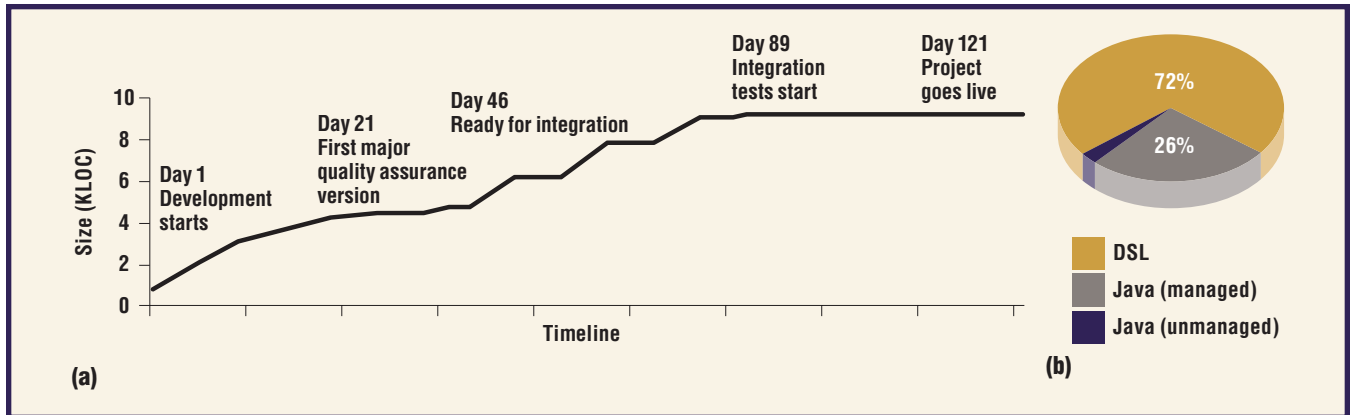


Figure 8. A case study of a customization project from kickoff to deployment: (a) the growth of customization code; (b) breakdown into kinds of code. Once an approach such as ModelTalk is in place, the organizational benefit is clear: a short time to market.

About the Authors



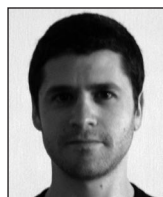
Atzmon Hen-Tov is the chief software architect at Pontis and has led the development of ModelTalk since its inception. He's interested in model-driven development and other methods for software development industrialization. Hen-Tov is a member of the ACM and the IEEE. Contact him at atzmon@ieee.org.

David H. Lorenz is an associate professor in the Department of Mathematics and Computer Science at the Open University of Israel. His research interests include aspect-oriented software engineering and programming, particularly involving multiple domain-specific languages. Lorenz has a PhD in computer science from the Technion-Israel Institute of Technology. He's a member of the ACM and the IEEE. Contact him at lorenz@openu.ac.il.



Assaf Pinhasi is a senior software architect at Pontis. His interests include domain-specific languages, Web technologies, and scalable software architecture. Pinhasi has a BSc in computer science and mathematics from Tel-Aviv University. Contact him at assafpinhasi@yahoo.com.

Lior Schachter is a senior software architect at Pontis and head of the ModelTalk Core team. His interests include model-driven development, domain-specific languages, and adaptive systems. Schachter has a BSc in electrical engineering and computer science from Tel-Aviv University. He's a member of the IEEE. Contact him at liors@ieee.org.



Acknowledgments

We thank Zvi Ravia and the anonymous reviewers for their helpful comments on the article. ModelTalk was influenced by the TGP (Type, Generic class, Profile) methodology developed by Shay Ben-Yehuda. We're grateful to Pontis, a leading vendor of online marketing automation solutions for the telecommunications market, for granting us access to their development process and data. This work was supported partly by the Israel Science Foundation under grant 926/08 and by the office of the chief scientist of the Israel Ministry of Industry Trade and Labor.

References

1. P. Clements and L. Northrop, *Software Product Lines—Practices and Patterns*, Addison-Wesley, 2001.
2. M. Voelter and T. Stahl, *Model-Driven Software Development: Technology, Engineering Management*, John Wiley & Sons, 2006.
3. D.S. Batory et al., "Achieving Extensibility through Product Lines and Domain-Specific Languages: A Case Study," *Software Reuse: Advances in Software Reusability, Proc. 6th Int'l Conf. (ICSR 00)*, LNCS 1844, Springer, 2000, pp. 117–136.
4. A. Hen-Tov, D.H. Lorenz, and L. Schachter, "ModelTalk: A Framework for Developing Domain-Specific Executable Models," *Proc. 8th Ann. OOPSLA Workshop Domain-Specific Modeling (DSM 08)*, ACM Press, 2008; www.dsmforum.org/events/DSM08.
5. M. Fowler, "Language Workbenches: The Killer-App for Domain-Specific Languages?" 2005; <http://martinfowler.com/articles/languageWorkbench.html>.
6. M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern," 2004; <http://martinfowler.com/articles/injection.html>.
7. D.H. Lorenz and J. Vlissides, "Pluggable Reflection: Decoupling Meta-interface and Implementation," *Proc. 25th Int'l Conf. Software Eng. (ICSE 03)*, IEEE CS Press, 2003, pp. 3–13.
8. R. Razavi et al., "Language Support for Adaptive Object-Models Using Metaclasses," *Computer Languages, Systems and Structures*, vol. 31, nos. 3–4, 2005, pp. 188–218.
9. J.P. Briot and P. Cointe, "Programming with Explicit Metaclasses in Smalltalk-80," *ACM SIGPLAN Notices*, vol. 24, no. 10, 1989, pp. 84–96.
10. B. Selic, "A Systematic Approach to Domain-Specific Language Design Using UML," *Proc. 10th IEEE Int'l Symp. Object and Component-Oriented Real-Time Distributed Computing*, IEEE CS Press, 2007, pp. 2–9.