A Unified Object Topology

WILLIAM M. TEPFENHART and JAMES J. CUSICK, AT&T

To navigate the object, pattern, and architecture fields, the authors developed the Unified Object Topology, which uses a technology's domain dependency and implementation detail to organize relationships with other technologies and identify how the system will evolve. It also supports object repositories and identifies future research directions.

Object-oriented, patterns, and architectures provide varied, rich, and often challenging technical terrain for software developers, managers, and researchers. However, using these technologies does not guarantee that your software project will succeed. Within the object technology terrain lie rich valleys of well-proven technologies. There are also barren plains and frigid peaks of poorly understood techniques and methods. To find the paths between fruitful valleys you need a map or an experienced guide.

Our Unified Object Topology provides such a map. It offers a cohesive view of several separate but related technologies and techniques, and lets you locate major technologies and their interrelationships in today's complex development environments.

Figure 1 shows the composition of the topology, which relates frameworks, kits, object patterns, domain models, architectural styles, and domain taxonomies. By sequentially following the conceptual paths between technologies, you can more clearly design your products. The topology also provides a context in which you can better use these technologies in large-scale distributed systems.
THE TOPOLOGY

In our careers thus far, we have served as consultants, university instructors, system architects, authors, programmers, and researchers. These experiences raised questions of how all the technologies we were using were related, and what the next step would be in synthesizing the myriad and powerful software abstractions and components that lay before us. Our answer was to develop the Unified Object Topology, which not only organizes concepts and tools, but points to new and relevant questions, many of which we have yet to answer.

Map making. A topology, or map, can have multiple dimensions. We are concentrating here on a 2D view of object technologies, but we have also developed topologies for relational technology. As Figure 1 shows, the topology lies on a grid of common attributes that reflect the technologies being mapped. Using the attributes of Implementation and Domain as defining points, each technology falls within a specific region of the grid.

The implementation attribute reflects how a software artifact is expressed (for example, in drawings, text, or source code). Development endeavors expressed in natural language or graphics would have an Abstract implementation attribute. On the other hand, if the element is expressed in machine-executable form, then the implementation attribute would be Concrete. If we use formal description languages to map an abstract description into a concrete description, the range of these values is continuous.

The domain-dependency attribute describes development topics in relation to the application domain. If the topic is described using domain terms, it is considered detailed; if not, it is considered domain-independent. Because descriptions can range from domain-dependent to domain-independent, the range on the domain attribute is also continuous.

Technology terrain. Armed with the appropriate topological canvas, we measured and distributed the major object technologies in relation to the implementation and domain-dependence attributes. For our analysis, we focused on each technology's relation to the axis and to each other; the shape and size of each technology is only suggestive.

This gives us a global view of, for example, what makes an inventory control system developed for a retail chain distinct from one developed for a petrochemical plant. As shown in the lower right of Figure 1, domain technologies are tightly bound to the application domain, but are placed far into the abstract-implementation range because of their graphical and natural language implementation.

- Domain models are also in the lower right of Figure 1. Dozens of methodologies offer techniques on how to derive such models. However, the analytical representations that result remain largely nonexecutable, yet tightly bound to the application domain. Within the telecommunications systems domain, for example, we have seen models with practically no domain overlap because of the high degree of specialization in such systems.

- Architectural styles are sets of operational characteristics that identify an architectural family. Styles such as on-line transaction processing, data-streaming systems, decision support systems, and real-time systems are domain independent. For example, decision support systems can be built for financial applications and for combat operations. Each system can be described in highly abstract terms on both the implementation and domain continua. In other words, you cannot compile a box diagram, but it may serve multiple domains well.

- Frameworks are sets of connected, cooperating classes that make up a reusable design for a specific software class, providing the entire domain-independent infrastructure you need to implement an application. The most common commercial frameworks today implement graphical user interfaces. At one level, GUI frameworks reside in the computer-interface application domain. Realistically, however, they float free as part of the infrastructure that lets a business application take form. Frameworks are thus independent of
the application domain but nearly concrete in implementation.

- **Kits** are logically related objects that support a single application concept. They have high domain detail and fairly concrete implementations. Kits differ from frameworks in that a kit's objects are physically independent but domain dependent. Thus, a class family can be constructed as a kit or a framework, depending on how closely bound the class family is to a single domain. We've found this to be a better classification scheme than others, such as the number of classes in a class family.

- **Object design patterns** are predefined design structures used as building blocks for a software architecture. The part-whole pattern, for example, says that some objects are part of another object, which is then considered the whole. This design pattern can be applied to a car and its parts just as easily as to a corporation and its parts. Patterns can also be classified on three levels of granularity: architectural frameworks, design patterns, and idioms. As a result, patterns in the topology stretch from somewhat detailed to independent in domain, but are limited to a narrow, abstract dimension of implementation.

- **Applications** are the implemented, domain-specific, mostly executable development artifacts. We build them from all the other technologies and abstractions.

**TOPOLOGY AT WORK**

Now that we have a map, we can pick a destination and navigate among the contours. Using the technologies and the relationships inherent among them, dynamic paths emerge in the topology that can help you:

- direct software systems construction,
- develop reusable components,
- manage object repositories, and
- direct research efforts.

When you develop a large-scale system, you face many problems at once. You must analyze the application domain and define an architecture. You must map the required functionalities against the individual processes. You must implement the domain classes and hope, along with everyone else, that you can create a cohesive system.

You can view this balancing act between multiple development tasks and the technologies that support them as a traversal of unique paths through the topology. These development tasks result in frameworks, kits, and system implementations.

**Framework development.** The upper arch in Figure 2 illustrates the path for developing a new framework. You must first identify the framework using the architectural styles. You then use the object patterns to identify the application elements (of a particular architectural style) independent of application domain.

A framework built in this way will be easier to understand and have better performance characteristics because it arises from a single, well-understood architectural style rather than a mixture of architectural elements. Class names and relationships also reflect a particular architectural style. For example, a data stream framework only includes classes relevant to a data stream infrastructure and does not carry the overhead associated with unnecessary capabilities such as GUI classes and event-handling code. Using different frameworks to connect subsystems will also be easier. Finally, integration efforts benefit from interframework connections, which you can develop independent of specific applications.

**Kit development.** The lower arch in Figure 2 illustrates the path for developing a new kit. You first identify a domain using the domain characterization. You then design the kit's classes using the object patterns, which identify the relationships among the elements specified by the domain taxonomy. This is done independent of the architectures or frameworks in which the classes will be used.

Kits developed using this process will have much greater reuse potential than those developed otherwise; once you tie a class definition to a particular framework, you can only use it in that context. Classes within a kit will undoubtedly be more complex because they...
reflect a much broader base of use. However, over time, the return on the increased resources required to develop architecture- and framework-independent kits will be much greater than that of current practices.

System development. The development arc of a large-scale system traverses the topology, as Figure 3 shows. To identify the problem, you apply the domain taxonomy and domain characterization. You then lay out the basic system architecture using the architectural styles. You use the frameworks to design (and implement) the individual processes. Finally, you add required domain-specific processing by selecting the appropriate kits and assembling them using design pattern guidance.

Following this sequence—and given a sufficient number of frameworks and kits—you reduce actual development (code writing) to linking the classes within the kit to the classes within the framework. The idea here is to develop as little complex, application-specific code as possible.

Once an application is complete, you can use the verification and validation process to further refine and develop the domain taxonomy. That is, if the system is not meeting customer needs, the problem could lie in the domain taxonomy, domain model, architectural style, or framework. The dashed arrow in Figure 3 shows that our initial problem classification is correct; it may also show that our taxonomy needs further refinement if the application doesn’t fit in the existing taxonomy. The upward branch shows that our analytic model description works when implemented, and thus we understood the problem correctly.

Commercializing software resources. The topology can also identify gaps in the software market. The shaded sections in Figure 4 show the commercial or public availability of various technologies based on our market research estimates. Although some products are available in nearly all Unified Object Topology categories, each still shows unmet demand.

Although there are many frameworks on the market, most are for GUI front-end systems. On the other hand, there are virtually no commercial frameworks for architectural styles. For example, there are virtually no frame-

Organizing object repositories. Our topology also crafts the foundations that let you organize, populate, and maintain object repositories. For example, the mechanisms you use to access objects can be organized by domain taxonomy for kit classes or by architectural styles for framework classes. Repositories can also be grouped by framework, organized by architectural style, and by kits, organized by domain taxonomy.

Using the topology, you can evaluate a given class family (whether externally or internally developed) by whether it fits into a framework or kit category. If the domain and the application infrastructure are tightly coupled in a class family, it won’t fit nicely in either category, thereby limiting its reusability and disqualifying it as a repository candidate.

RESEARCH DIRECTIONS

Many researchers are moving away from conventional computer science topics toward more application-specific topics. This gives rise to new problems that our topology can help solve.

Characterizing problem domains for application development is extremely complex and has much in common with object modeling in OO development and the conceptual analysis of database modeling. At present, we cannot clearly and unambiguously describe domains. The clear separation of domain models, architectures, and frameworks in the topology allows for further definition of techniques within each area and clear
methods for connecting them. We can focus on creating rich languages for describing domains.

We must also better understand the architectures of large-scale distributed systems. Many systems today are distributed and have advanced architectures, yet we cannot clearly and adequately describe these highly complex systems. The paucity of architectural styles underlines this issue. Our topology plots the technologies that require detailed investigation and shows how they are functionally related.

Our topology goes far to integrate the varied technologies of objects, patterns, and architectures, as well as domain models, domain taxonomies, frameworks, and kits. As we continue to survey their interactions, our understanding of these approaches will grow.

We view our topology as similar to the products of early European cartographers attempting to chart the New World: it is rough around the edges, somewhat errant in latitude and longitude, but dead-on in compass heading. With such early maps, people found their way to many new lands. With this topology, we have reached a deeper understanding of the software field today. We expect it to help us discover more technologies in the future.

ACKNOWLEDGMENT

We are indebted to the many reviewers of this article. Their astute criticisms contributed immeasurably to the development of our ideas.

REFERENCES


William Tepfenhart is a senior technical staff member at AT&T, where he specializes in technologies and architectures for telecommunications applications. He has worked previously at Knowledge Systems Corp., where he specialized in artificial intelligence applications for military programs, and in the Missiles Division at LTV Aerospace and Defense, where he applied AI to manufacturing and intelligent design.

Tepfenhart received a PhD in physics from the University of Texas, Dallas, in 1987.

James Cusick is a technical staff member at AT&T's Business Markets Division, where he researches and recommends software development tools and technologies for use across AT&T. Cusick also lectures on software engineering at Columbia University's Computer Technologies and Applications program.

Cusick received a BA in organizational psychology from the University of California, Santa Barbara, and a certificate in software development from Columbia University. He is a member of the IEEE Computer Society.

Address questions about this article to Tepfenhart at AT&T, HR 7A-023, 480 Red Hill Rd., Middletown, NJ 07748; william.tepfenhart@att.com, or Cusick at AT&T, Rm 14034, 55 Corporate Dr., Bridgewater, NJ 08807; james.cusick@att.com.

IEEE SOFTWARE