Product family engineering permits software component reuse, which saves both time and money. To succeed, the process must support variability among the family’s systems. The authors developed a simple method that uses patterns to model variability and applied it to development of spacecraft mission-planning systems.

For each of its spacecraft, the European Space Operations Centre (ESOC) develops a new mission-planning software system. MPSs are increasing in complexity, even as tighter budgets require greater efficiency in mission development and operations. This led ESOC to consider developing spacecraft MPSs that can be customized across a range of missions. ESOC’s approach to this is to develop a product family of spacecraft MPSs.

A product family is a group of similar products within a market segment, such as mobile phones, pensions, or spacecraft MPSs. Product-family engineering can generate significant savings in cost and time by permitting software-component reuse. But to succeed, product family engineering must provide mechanisms to support variability among the family’s systems. This variability permits a product family to evolve and maintain its relevancy.

We have developed a simple method that uses patterns to model variability. We start by analyzing existing user requirements from systems within the family and identifying discriminants. We then build an object-oriented family model, using a set of predefined patterns to model the discriminants. This model lets developers identify and select desired features and build new family systems.
We tested our approach on ESOC’s spacecraft MPSs. The MPS family provides a good test application for our method because its systems vary widely and it is representative of a typical product family. Also, ESOC’s previous attempts to build a generic mission-planning system were unsuccessful.

**REUSE IN PRODUCT FAMILIES**

In product family development, reuse approaches range from code component libraries to generic-system parameterization. The library approach typically yields between 10 to 30 percent reuse. Its effectiveness is limited by several factors. First, behavior must be well-defined but general, which restricts component size. Second, engineers tend to suffer from “not-invented-here syndrome” and distrust work done elsewhere. Finally, code component libraries preclude analysis, design, and test, which constitute a large part of software project costs.

The generic-system approach can yield 100 percent reuse, but its effectiveness is limited because, as the number of family products increases, the number of common features decreases. For example, in a family of four systems, A, B, C, and D, there often exist common features between A and B, B and C, and A, B, and D, but rarely is one common to all systems. Customization facilities for ready-made products must change each time a new system is added to the family. Such facilities often become complex and difficult to use, increasing project costs and reducing the original savings.

Product family engineering offers an alternative reuse approach. Reusable components are built separately from the systems that will use them. Separating the intense and expensive development of reusable components—the “family engineering”—from the actual systems engineering makes the latter process both faster and cheaper.

There are four basic steps to family engineering: requirement definition, analysis, design, and implementation. Requirement definitions are gathered based on both domain expertise and the combined requirements of existing systems. The family requirements are then input to the family analysis phase, which generates an object-oriented model consisting of classes, objects, and patterns.

In the family design phase, the object-oriented model is extended to include solution-specific classes, such as user-interface elements, dates, and persistence. Finally, during implementation, the family model is mapped to a set of classes defined in the implementation language.

During systems engineering, analysis and design of the new system is generated by filtering out family classes and objects not traced to the system-requirements subset. After this filtering, the new system can be extended to include items that were consciously excluded at the family level.

**MODELING WITH PATTERNS**

There are several existing approaches to modeling variation and similarity in family systems, as described in the boxed text, “Approaches to Family Model Development.” Our approach models variation using patterns.

Patterns provide reusable, routine solutions to certain types of problems and support the reuse of underlying implementations. Among their advantages:

- permit domain-variations modeling using a single technique,
- support modeling of large, complex systems, and
- make mapping from requirements to implementation directly visible.

In an object-oriented development environment, a pattern is typically a class diagram and an object diagram, but it might also include an object-interaction diagram and a state-transition diagram.

**Discriminants**

A discriminant is any feature (requirement) that differentiates one system from another. We use discriminants and their associated patterns to produce a unified family model that includes all commonality and variation across the family. These patterns can be incorporated into any architectural style and will let developers build product-family architectures, from which they can derive a new system’s architecture.

We use the family user-requirement specifica-
tion to identify discriminants, which come in three basic types:

- **Single discriminants** are a set of mutually exclusive features, only one of which can be used in a system. For example, all mobile phones have a display, but displays can vary, such as by the number of displayable characters.

- **Multiple discriminants** are a set of optional features that are not mutually exclusive; at least one must be used. For example, there must be at least one way of placing a call on mobile phones, but there can be several, such as pressing the digits, pressing redial, or voice dialing.

- **Option discriminants** are single optional features that might or might not be used. For example, mobile phones can have an Internet connection facility, but they do not require one.

Discriminants are always related to other requirements. For example, the option-discriminant Internet connection facility is coupled with the feature's particular requirements.

Our discriminant types closely resemble the mandatory, alternative, and optional feature properties identified in the feature-oriented domain analysis described in the boxed text, “Approaches to Family Model Development.” However, our multiple discriminant differs from FODA’s alternative feature. Further research will be required to demonstrate whether our discriminants are necessary and sufficient to represent variability.

**Modeling discriminant types**

During family analysis, we use the family user-requirement specification to build an architecture that will support all family systems. This architecture must be independent of all discriminant states. For example, the architecture must be valid whether an optional feature is included or not. To achieve this independence, we use patterns to factor out and isolate each of the discriminants. We have defined three patterns to model the three discriminant types. Our object and class diagrams use an extended form of OMT; a similar extension of UML is also possible.

**Single adapter pattern**

The single discriminant can be modeled as an inheritance hierarchy in which generic features are modeled in a base class and specific features are modeled as subclasses. Only one subclass can be instantiated in any single system. The set of subclasses is known as a realm. If required, the realm can be extended for a new system. Figure 1 shows a single adapter pattern. The dotted box denotes a realm; the three dots within it indicate that the realm can be extended in the future.

Sometimes, methods in a subclass must be accessible to other classes in the model. In such cases, we place virtual functions in the base class—Operation B in Figure 1—so other model parts can refer to base-class instances without knowing which subclass a given system will use. We achieve this by creating a single subclass instance using a well-known name; that is, we create a variable with an identifier that other developers will recognize. We implement this single instance using the singleton pattern. A subclass must not introduce new methods, because this would preclude its replacement by another subclass.

**Multiple adapter pattern**

We model the multiple discriminant in the same way as a single discriminant: as an inheritance hierarchy, with generic features modeled in a base class and specific features modeled as subclasses. However, in this pattern, more than one subclass can be instantiated in any single system. Figure 2 shows a multiple adapter pattern in which each subclass has
one instance and the set of instances is held in a collection.

The problem with having an indeterminate number of subclasses is that they are difficult to access from the rest of the architecture. Although the subclass methods are defined by the base class, we cannot determine the actual instances until a system is built. To access methods in a particular subclass, other model parts can refer to a base-class instance without having to know which subclass a given system will use at a given time. We achieve this by identifying the subclass instances by a name string or some other unique identifier and storing them in a collection. Binding the rest of the system to the subclass is thus a runtime operation. Finally, each new system must provide instances of the subclasses and add them to the collection.

Option pattern

When a feature is optional, we model it by creating two associated peer classes. We do not model optionality as inheritance, because inheritance cannot be optional. The associated classes must have a 0-1 relationship on at least one end. Figure 3 shows an example: Class B is an optional class with Class A; Class A does not assume Class B exists, and therefore can be reused whether or not Class B is reused.

This pattern can also be applied to “related-to” and “aggregation” associations. For example, a person who might or might not have a mobile phone would be an optional related-to association. A mobile phone that might or might not have a built-in modem is an example of an optional aggregation association.

Finally, discriminant types are sometimes combined, in which case we merge discriminant patterns. If we had a set of features that are optional but not mutually exclusive, for example, we would model this using an option pattern combined with a multiple adapter pattern. With e-mail on a mobile phone, for example, the e-mail itself is an option and the e-mail delivery protocols are a multiple discriminant.

**Spacecraft Mission-Planning Systems**

ESOC uses spacecraft mission-control systems to monitor and control satellites. The mission-control system works by transmitting a sequence of operations to the satellite and monitoring the telemetry the satellite sends back. Mission plans carry the sequence of operations. These plans are based on requests from both spacecraft-control engineers and customers, such as weather forecasters. The mission plans are generated by a mission-planning system, or MPS, and then edited by an MPS staff member before being sent to mission control.

Each MPS requires about 10 person-years of development effort. ESOC’s attempts to build generic MPSs have been unsuccessful. Although each system’s high-level functionality is the same, the details of each spacecraft and its orbit have made reuse very difficult.

**Defining MPS requirements**

We built a family user-requirement specification by editing and merging the requirement specifications from three separate MPSs: ISO, a spacecraft that observes stars; ERS-2, a remote-sensing spacecraft that monitors the earth’s environment; and Cluster, a multispacecraft mission to monitor the earth’s magnetosphere. We selected these systems because they represent a wide range of product family functionality and are thus a good starting point for building a product family model and architecture.

Each MPS requirement specification had about 150 user requirements. Our family user-requirement specification had 350 requirements. Here are examples of requirements:

- It shall be possible to enable plan optimization for all of the plan.
- It shall be possible for several alternative plans for the same planning period to reside in the system simultaneously.

Once gathered, we entered the family user requirements into a relational database using Microsoft Access. Next, as Table 1 shows, we identified 10
discriminants and their corresponding patterns. Each discriminant is fundamental to MPSs; together the 10 represent large variations in functionality.

Our analysis of the MPS family produced 20 class diagrams, 15 object-interaction diagrams, and 100 classes. Following are more detailed examples from our work.

Example: single adapter pattern

With any family model, we must factor in changes in user-interface technology. Figure 4 shows a base class, UserInterface, and a realm, which includes two subclasses: a Motif user interface and a Windows user interface. In product family development, developers often have to write user-interface access code before they know which type of interface will be used. Thus, as Figure 4 shows, we create an object, which holds the UserInterface instance and lets other system parts use it without knowing what it is. This lets us postpone actual user-interface instantiation until a specific system is built. We do this in C++; other languages support similar approaches.

Example: multiple adapter pattern

MPSs use many different planning methods, which are in turn used at different times during the life of the system. Figure 5 shows a base class, Planner, and a realm that includes three subclasses: a first-come, first-served planner; a priority-based planner; and a planner that attempts to minimize power consumption. The figure also shows the inclusion of a Collection object. The Collection object holds the planner instances, letting other system parts access them without knowing their nature. This is important as, when other product family components are being developed, developers cannot assume any particular planner subclass or number of planners. We deal with this in C++ by storing the planners in a collection and using identifiers to access them.

Example: option pattern

The number of events logged in each new system in the family vary. For example, the fact that the user interface may or may not have an event log is a discriminant. Figure 6 shows the UserInterface class and the EventLog class. In an actual system, there would be one instance of each. Because the event log is an optional feature, developers cannot assume that it will exist. We thus define a static member function for EventLog, which returns the address of an instance and finds instance. If no instance exists, it will return a null pointer.

DISCUSSION

It took the two of us 12 person-weeks to produce the MPS family analysis model. Of this, we spent five person-weeks consulting domain experts and reviewing existing documentation. In the remaining seven person-weeks, we created and reviewed the model. The actual project duration was six months.
due to the limited availability of domain experts. Our family analysis encapsulated all of the MPS domain variability in a single unified family model without restricting family flexibility. Identifying broad differences between family products is straightforward; the difficulty comes in finding agreement on the details of distinguishing features. This is true for several reasons. First, not every domain expert is familiar with every product in the family. Second, each expert might have a different opinion of why products have certain distinguishing features. For example, experts might identify a different separation point of generic from specific, which is critical to discriminant identification. Third, subtle product differences in ontology often hinder discussion and negotiation.

Our method assumes that existing requirement specifications are available and have numbered atomic requirements. In ESOC’s development environment, the documentation satisfied this assumption. Legacy system documentation not written in this way would require additional effort to isolate requirements for reuse.

**Future work**

At this point, we have only developed patterns that deal with variability in structural rather than behavioral functionality. Further work must be done to identify patterns to model variability in behavioral functionality. This is particularly important in embedded system applications, such as spacecraft MPSs. Also, thus far we have only undertaken family analysis; family design will extend our output to design solution-specific patterns and classes such as user-interface elements, dates, and persistence. Also, sometimes there is more than one design for the same analysis. Future work must confirm whether the variability types we identified hold true for design.

Once the family design is complete, the decision about which components to build is a commercial one. Factors to consider include:

- development costs,
- how many times the component will likely be used,
- the cost of providing the functionality without a reusable component, and
- the priority of other characteristics, such as added reliability and reduced time to market.

**Evaluating the costs and benefits of our approach is difficult.** The quality of any product family model depends on more than one technique. The true benefits can only be determined when a set of components have been built and at least one new system in the family has been developed using the architecture and components. However, our method simplifies the process of building complex models that support variability. We are now beginning a complete application of our method, which will include analysis, the development of architecture components, and system implementation.

**Acknowledgments**

We thank all the reviewers of this paper for their helpful comments. Their astute observations contributed considerably to the development of our ideas. This article is part of collaborative research between the Napier University...
During family analysis, it’s important to develop a family model that captures the similarities and variations of family systems. Several approaches toward this end have been suggested. For example, David Parnas suggests three methods for supporting code-level variability, while Will Tracz suggests a domain-specific “generic” architecture that captures common family features but ignores variations. Hassan Gomaa suggests modeling different types of variation using different techniques, including aggregation hierarchy, generalization and specialization hierarchy, and feature and object dependency. However, integrating such techniques is difficult, and the roadmap from problem to solution is hard to follow.

Another method is feature-oriented domain analysis (FODA), in which feature variability is modeled during the domain-analysis phase. However, unlike our pattern method, FODA does not provide mechanisms for modeling variability at the architectural level and thus cannot derive a single product-family architecture from the overall product family architecture.

REFERENCES