17

Telvent

with Jesús Bermejo
Pablo Trinidad
David Benavides
Antonio Ruiz-Cortés

Company facts of Telvent

Organisational size: >1,000 developers.
Starting Mode: A configurable product for many clients with changing requirements.

Experienced improvements:
- Server platform extended to other markets.
- Introduction of run-time variability.
- Improved reference process framework.
- Centralised roadmaps for platforms.
- Market platform in a different domain.

Business: Alignment of strategy and architecture.

Architecture: Use of dynamic abstract factory pattern in the platform.

Process: Improved framework.

Organisation: Separation of domain and application engineering organisations.

17.1 Introduction

Telvent is specialised in solutions in four specific industrial sectors: energy, traffic, transport and environment. Its main clients are in the Americas, Spain and China. With over forty years of experience in industrial supervisory control and business process management systems, Telvent executes projects and provides technical services in the field of mission-critical, real-time control and information management. Telvent provides outsourcing and consulting services, and employs a technology-neutral philosophy. The company manages IT and telecommunications infrastructure for an extensive international client base.

17.2 Motivation

The development of software for control, supervision and management falls within the category of complex system engineering. This type of software deals with strong non-functional requirements such as time responsiveness to accomplish real-time requirements for the control of complex systems, customisability to cover different cultural contexts and national standards, and maintainability. Often, critical systems are controlled, and upgrades must be performed rapidly.

Software product line engineering can be used to manage variants for common issues in the field of control software such as multiple communication protocols over the same channel, communications redundancy, extensive control at the remote terminal unit locations, remote configuration changes, data transfer to and from databases and fault tolerance in the context of a distributed architecture with support for a growing number of communication infrastructures. Many systems use predominantly long-distance communication, although short- distance communication may also be present.

This chapter summarises some aspects of software product line engineering at Telvent for one of its core business domains. It focuses on the conception phase of a product line targeting only a single product for which the requirements were expected to change widely. The experience shows how software product line engineering was successfully applied as a technique to enable the product to adapt to evolving requirements.

The customer asked for a real-time television software framework as part of the product. They wanted software that could capture a television signal from a card plugged into a PC and show the result on screen after applying some filtering and transformations. They did not really know which kind of filtering and transformations the framework had to support in the future. However, it was to be expected that after starting the development of the framework, the customer would want more functionality with similar or improved performance.

Late changes in requirements generally involve reduction of functionality and the loss of quality and time. In the situation at hand, many new requirements were expected to appear, and the architecture would have to be able to support them. Support for change had to be part of the software architecture.

Here, classical requirements analysis would leave a lot of black holes that would need to be solved before designing a flexible platform. Instead, software product line engineering was used to create a platform that supports the needed variability. Software product line development usually starts with domain analysis, where the main features of the platform are detected. In this case, the problem domain was thoroughly studied to define a product that would fit not only the (future) requirements of this first customer, but also the requirements of other companies in the same market with little effort.

To achieve this, the commonalities were analysed among the potential products that the different customers may request in the near future (cf. Fig. 17.1). The first customer wanted to compose television signals, bitmaps, videos and other images and apply all kinds of effects to them. The results of an analysis of existing commercial products in the real-time television market lead to additional requirements. Starting from this information, the common requirements of the platform were determined:

- The software should draw a final image as a result.
- Several sources of images or layers compose the final image, e.g. television signal capture, bitmaps, stored video, video streaming, text and 3D images.
- The layers overlap following a configurable order.
- Effects or transformations may be applied to one or more layers, e.g. black and white, and transparency effects.

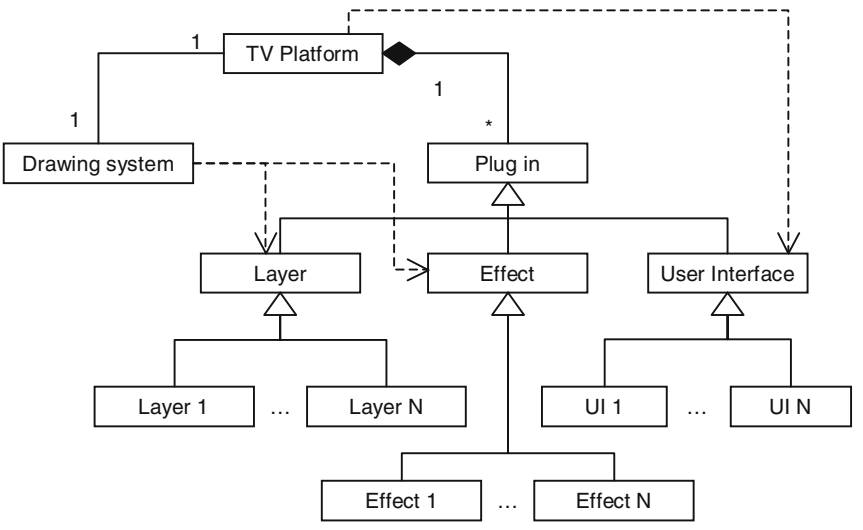

Fig. 17.1. Television software platform logical structure (UML)

- A user interface has to be provided to control layers and effects.
- It must work 24 hours, seven days a week. If used for television retransmission it must never stop working.

In short, the platform draws a sequence of layers, and optionally applies any of a range of filters to them. The platform has to support any kind of layers, effects and user interface. Updates should be supported, but the execution cannot not be stopped for maintenance.

The domain engineering team had to develop two major functional components:

1. *Drawing system*: in charge of drawing layers and effects in the television output.
2. *Plug-ins system*: because it was intended to be a 24/7 system, new versions and updating must be automatically installed at run-time without stopping the system. Layers, effects and user interfaces are considered plug-ins that easily connect to the drawing system.

The application engineering team was responsible for developing plug-ins for specific customers. These plug-ins could be promoted to the platform if more customers would demand them. In this case, the domain engineering would take over their maintenance.

17.3 Approach

The process for software product line engineering described in [106] was used. The following main sub-processes were distinguished:

- *Domain engineering* deals with core-asset development, where common features are developed.
- *Application engineering* deals with product development, where products are developed from common and specific features.
- *Co-ordination* deals with overall product line management, where synchronisation between the other two activities is arranged.

This is the basis to improve the development process and to support the needed variability. An important task is to decide which are the core-assets and which are the customer's product-specific features.

This section explains how the organisation was structured to deal with the business needs. Next, we explain how the domain engineering team dealt with the architecture. Especially, we consider the design of the plug-in system that may be used in other contexts. Its origin is an existing design pattern to automatically support the run-time connection of new plug-ins or components, i.e. layers, effects and user interfaces.

17.3.1 Organisation and Business

Following the process structure, development was separated in domain and application engineering groups. The domain engineering group was responsible for the platform development and the quality of the systems. The software product line infrastructure has to provide solutions not only for existing systems, but also for future systems. It has to deal with the rapidly evolving technological market nowadays. The domain architecture has to fulfil the derived business strategy requirements.

Domain engineering uses an architecture-centric approach driven by the business strategy. The technical solution is shaped according to long-term strategic and business objectives. Interfaces for software variants are tightly aligned with business variants. It is important to analyse them from both business and software perspectives. This keeps the strategy and planned evolution for the whole product line consistent with one another.

17.3.2 Using the Abstract Factory Pattern

The implementation of the product line depends on the Abstract Factory design pattern [59]. It can fulfil many common requirements of the plug-in system and provides fast development of systems in the product line.

Abstract Factory provides an interface for creating families of objects without knowing their concrete classes. Abstract Factory can be applied when a system has to be independent of how its products are created, composed and represented. This fits the requirements of the television framework, where layers, effects and user interfaces should be created independently of their concrete functionality, which varies from customer to customer. The design structure of this pattern can be seen in Fig. 17.2.

The participant classes in this pattern and their functionality are as follows:

- *Abstract Product* declares the interface for a type of product object. In this case, layers, effects and user interfaces will be abstract products.
- *Abstract Factory* declares an interface for the operations that create abstract products.
- *Concrete Product* implements the abstract product interface to define a concrete product that is created using a concrete factory. Examples of concrete products are a bitmap layer and a black and white effect.
- *Concrete Factory* implements the interface of the abstract factory. Each customer will require a set of layers, effects and user interfaces, and will have its own concrete factory implementation.
- *Client* uses the interfaces of abstract factory and product, but does not know about concrete implementations. A customer-dependent concrete factory will be instantiated beforehand.

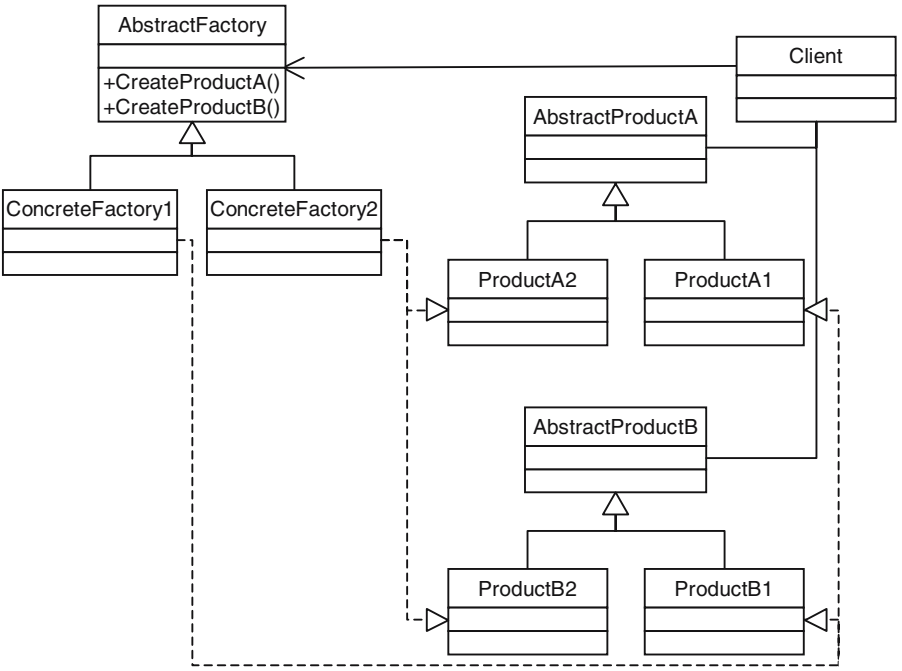

Fig. 17.2. Abstract Factory pattern (UML)

In the television framework, layers, effects and user interfaces are the abstract products to be created. The abstract factory is in charge of creating their implementations as Concrete Products. Each customer-specific product has a specific concrete factory (Fig. 17.3).

17.3.3 Introducing the Dynamic Abstract Factory Pattern

The Abstract Factory pattern has two limitations when it is used to implement variability:

1. Some concrete products are not pre-defined and should integrate into the application at run-time. Abstract Factory can only create concrete products if they are identified *a priori*.
2. A customer may solicit more than one instance of an abstract product. For instance, a customer may need a television capture layer and a bitmap layer at the same time. Abstract Factory allows only one concrete implementation per interface.

Considering these limitations some new features were added to the Abstract Factory pattern, thus creating a new pattern coined *Dynamic Abstract*

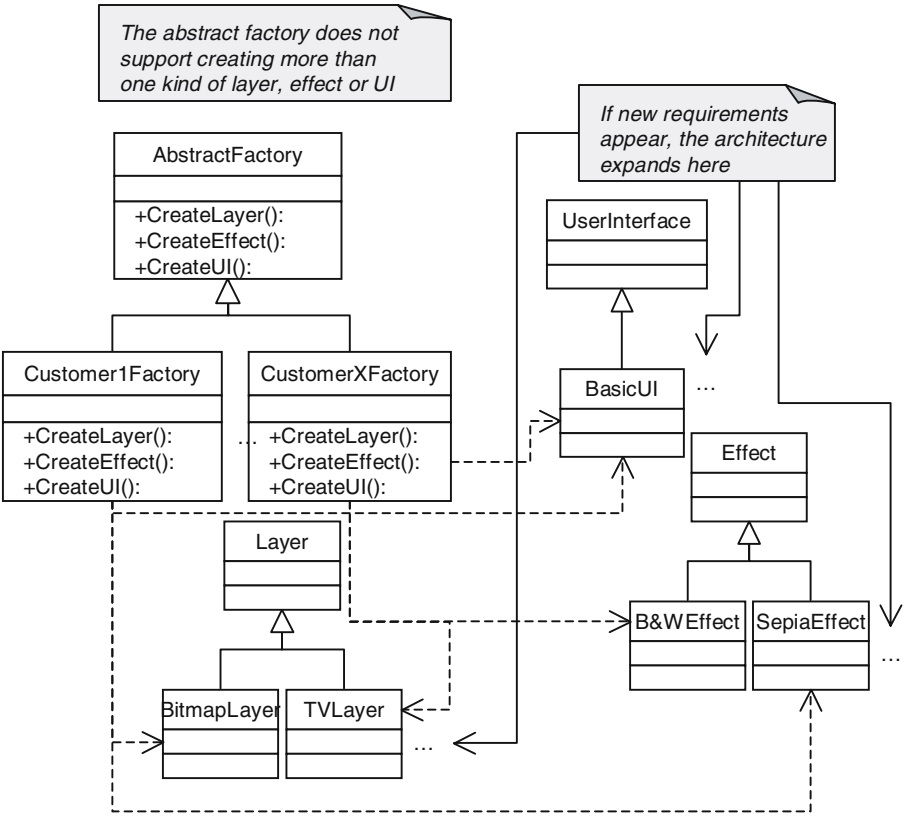

Fig. 17.3. Abstract Factory pattern applied to Telvent's platform (UML)

Factory. The new pattern is considered dynamic because it can change the relations between concrete factories and concrete products at run-time.

Dynamic Abstract Factory allows a concrete factory to create more than one instance of a concrete product. To this end, the methods that create concrete products, e.g. *CreateLayer*, were extended to receive a parameter that indicates which layer to create from all the available ones.

Furthermore, concrete factories were adapted to support new concrete products that were added at run-time. *Register* and *UnRegister* operations were added to Abstract Factory for each abstract product, e.g. *RegisterLayer* and *UnRegisterLayer*. These functions associate an identifier with a concrete product. This identifier is used when creating new instances of a concrete product.

For each customer, a set of layers, effects and user interfaces is available, and many others may be installed and used at run-time. Each customer hasa

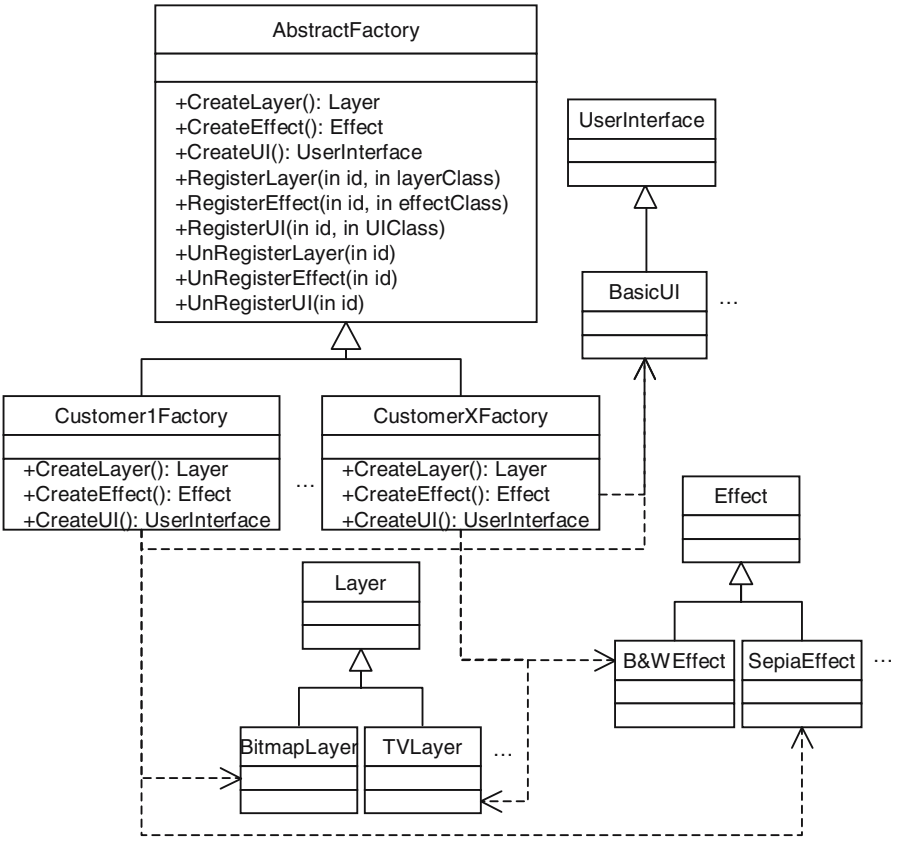

Fig. 17.4. Dynamic Abstract Factory pattern applied to Telvent's platform (UML)

concrete factory that initially registers the available concrete products. A UML model of the Dynamic Abstract Factory pattern is shown in Fig. 17.4.

17.3.4 Reusing the Dynamic Abstract Factory Pattern

To make the Dynamic Abstract Factory implementation reusable in other platforms where different abstract products are considered, some more adaptations were necessary. The *register* and *unregister* type of methods in Fig. 17.3 are not reusable because they are linked to a concrete context through their names. Instead of creating new methods for each new problem domain, a domain-independent method that supports the creation of any concrete product is better reusable. Consequently, a generic abstract factory was defined that creates only one kind of abstract product. The generic dynamic abstract

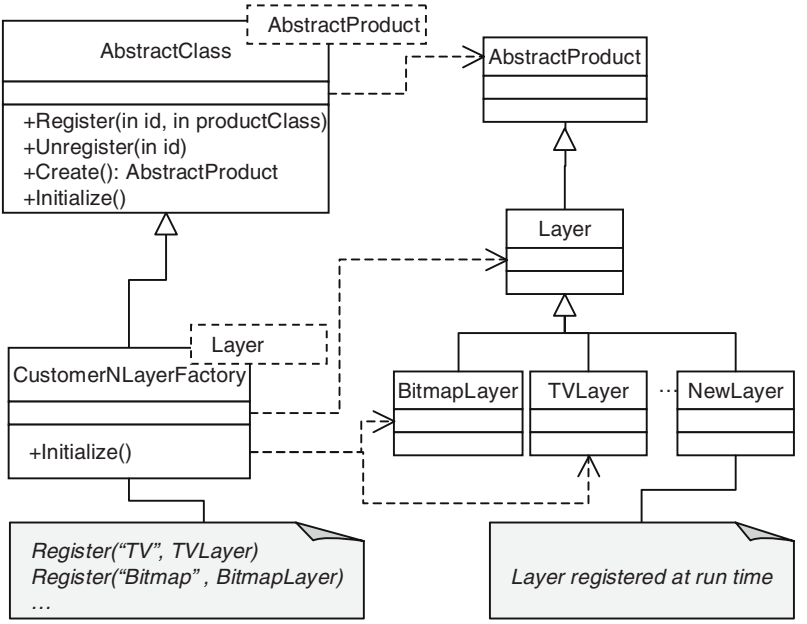

Fig. 17.5. Dynamic Abstract Factory pattern implementation using parameterised classes (UML)

factory designed was based on parameterised classes.[1] A UML model, representing this, is shown in Fig. 17.5. Applying this generic pattern to the television framework yields three concrete factories for layers, effects and user interfaces.

For each customer, the *Initialise* method is adapted and a set of concrete factories created. Support for integrating new concrete products comes from using the register method at run-time. But introducing new products means introducing new code. The insertion of code in a running process is not a trivial task. Depending on the language and environment, the dynamic loading of code can be achieved in different ways. In the television framework, dynamic libraries were used to plug and register new classes into a concrete factory at run-time. Each library contained one or more concrete products. When loading a library, the new concrete product is registered at a concrete factory. When unplugging that library, the concrete product is unregistered again.

The Dynamic Abstract Factory pattern is not linked to a concrete operating system or programming language. The solution is specified at design level, and can be applied using any programming language that allows the dynamic loading of code, for example Java or C#.

[1] A parameterised class is a class where one or more types are defined during instantiation. In C++, they are also known as *templates*

17.4 Lessons Learned

Software product line engineering can be applied in domains that are changing fast. In the case presented here, a television framework was set up as a software product line from the very start, when only a single customer was identified. This helped to solve three challenges:

1. The requirements of the initial customer were known to change over time.
2. The product was expected to be interesting to other customers in the same market as well, but development could not be delayed until those customers were identified.
3. The product should be applicable in similar, but different domains as well.

Carefully analysing and managing variability helped to develop an architecture that could deal with future changes in requirements and that was suitable for a broad range of customers.

In this case, software product line engineering proved to be a good way to align business goals and system architecture.

This chapter illustrates that product line engineering can be of interest even in the early development phases, when only a single customer is identified. A product line was set up to deal with the fact that the customer had unclear requirements that were expected to change in the future.