Patterns for Plug-Ins

Klaus Marquardt
c/o Dräger Medizintechnik GmbH, D-23542 Lübeck, Germany
Email: klaus.marquardt@draeger.com

Abstract

This pattern collection helps to define, implement and package Plug-Ins specific to a configurable application. Central patterns are the Plug-In and the Mutual Contract between Plug-In and the application. Organisational patterns accompanying the separate products complement and support the technical flexibility. In the last section, issues of creating and sharing objects used on both sides are treated.

Introduction

Software customers demand a high amount of adaptability and extensibility. Who would buy an Internet browser that can not be extended - to visualise images and movies of newly emerging, powerful standard format? Who would equip a laboratory with an automation system that can not be extended - to communicate with the chemical analysers of the next generation, or the current ones from the concurrent manufacturer?

The areas for extensible software applications are many. And beyond mere adaptability users demand comfort. Extensions must fit seamlessly into the existing system, providing full functionality and exhibiting the most recent look and feel. Browsers that open a document hide the new-ness of its format; graphical load lists for the new chemical analyser look like described in its user manual.

The increasing demand for functionality not only leads to large software systems that require a strict dependency oriented decomposition and management. The users really expect todays products to support and integrate tomorrows technology - to lengthen the product life time and raise return on investment. While this expectation matches fine with promises of object oriented techniques, building these products and product lines requires more than virtual functions.

In the past, software engineers have tried to rely on existing standards and define missing ones. The emerging „component based development" shows that these standards lack completeness. Most standards on network protocols are fundamental to the importance and success of software today, more network standards are under construction. But the standards that try more than connecting, that define an application layer with a domain specific „semantic context", live in niche markets and are often considered insufficient.

So what is missing? There are often specifics to be covered. No application domain is sufficiently covered by the above approaches; the most important reason for this being that no domain is really closed. Avoiding the shortcomings of standard - this is what the patterns here deal with.

The patterns are presented in three sections. The first section, „The Application and its Plug-In", defines what a Plug-In is, and focuses on what the application developers and architects have to care for to define the Plug-In. Section two, „Organisational Issues", shows Application and Plug-Ins can be developed together, how the projects must be separated or may interfere. Section three, „Creation and Factories", helps with technical details of classes and objects that are used by both the application and the Plug-Ins.

The patterns usage is illustrated by the fictitious ARGUS example. The security system ARGUS monitors a building for violation of access rights, temperature limits, and possibly many others. All violations are reported to a workstation both visibly and audibly, depending on their priority. Additionally, a town central OLYMP can connect to a large number of individual security systems, and observe all violations of all connected buildings.

Copyright © 1999 by Klaus Marquardt. Permission is granted for use with PLoP conference series.

Section I: The Application and its Plug-In

Pattern 1: Plug-In

Context

An application that is required to be highly adaptable, or be extensible to support future functionality or modules.

Problem

How can functionality be added late? How can the functionality be increased after shipping?

Forces

- At shipping time of the application, not all functional components are known or available
- The application must not presume that a particular Plug-In is available
- Early delivery increases market share and profit
- Kind of additional functionality is well known
- Specifically added functionality can not be foreseen
- Which functionality is dynamically added when, is determined at run time and can hardly be foreseen
- A technology evolves, the application will be used in unforeseen ways
- Shipping is expensive
- Application is not affected by additional functionality

Solution

Factor out functionality, and place it in a separate component that is activated at run time. This component is called a Plug-In. The application defines functionality that it does not provide itself, but must be added by Plug-Ins. The application is shipped with a well defined interface for Plug-Ins.

A Plug-In consists of executable code that the application loads dynamically at run time. Each Plug-In complies with the defined interface. The application does neither depend on a Plug-In internals, nor on the presence of a particular Plug-In.

To identify functions that can be placed into a Plug-In, look out for open points in the application requirements. Frequently, a specific kind of extensibility is required, or implied by „…“ phrases. Multiple subclasses of key abstraction are also candidates, if your analysis shows that extending the system would add another subclass.

Consequences

☺ Functionality can be developed and added after shipping the application

☺ Application with factored functionality can be shipped earlier than full functional application

☺ Occasionally, an application with defined Plug-Ins can be shipped whilst a full functional application could never be shipped at all

- ☺ Application is not updated when adding functionality, and is not affected by a Plug-In.
- 😐 Each Plug-In must be shipped separately, but can also be sold separately.
- ☹ The kind of extensibility must be foreseen, as the interface for Plug-Ins must be defined in advance

Implementation

Clearly separate between physical design (execution) and logical design (basic and added functionality). Physical design is up to the application, that decides when which Plug-In is activated in which process; for Plug-In internals, often a resource budget is defined as part of the interface. The application also defines the outline of the logical design, but internals of the subclass and its helpers are completely up to the Plug-In.

Plug-In can be implemented as DLL or run time library, or as active object (e.g. Active-X). The application decides about the activation time and conditions; in this respect a Plug-In is no application on its own. Nevertheless a Plug-In may start and use helper applications when useful.

When the functionality of the Plug-In is central for the user, the application serves as a starter and integrator for Plug-Ins. It must then be shipped with at least one Plug-In.

Variants[1]: Some applications do not make any sense without at least one active Plug-In. Some applications define more than one Plug-In interface, and expect different kinds of Plug-Ins simultaneously. Some applications allow only one active Plug-In at a time, others support an (almost) arbitrary number of different Plug-Ins of the same kind in parallel. Some applications even allow multiple Plug-In instances of identical type in parallel.

Example

The Olymp central can connect to a variety of different observation system. The specific, mostly proprietary transmission protocols are factored into Plug-Ins. This way the Olymp application is prepared to connect to a variety of different systems from different vendors. Each Plug-In is activated according to a schedule (when the corresponding local monitor becomes inactive).

Known Uses

The OpenCard standard defines the Plug-In interface that the Plug-Ins, Java Code on the OpenCard that is physically plugged into the system, comply with. Activation of the Plug-In is implicitly done at Plug-In time.

The Windows OS family has factored out the screen saver functionality, which must be provided by a separate Plug-In. Windows is shipped with a variety of different Plug-Ins; the user can select one of them to be activated.

ProductL has separated the analyser handling know-how and code into Plug-Ins. The user selects the kind and amount of analysers in the laboratory, and a corresponding number of the appropriate Plug-Ins is activated.

ProductW has separated the sensor and actor handling know-how and code into Plug-Ins. By physically attaching a sensor package, the appropriate Plug-In is selected and activated.

Netscape (like other browsers) factors out viewing functionality to support a variety of text and graphics formats.

---

[1] Introduction into terminology:
„Plug-In kind“ - different Plug-Ins are of one kind when they conform to the same predefined interface. Analogous to a base class.
„Plug-In type“ - the Plug-In code denotes the type. Analogous to a derived class.
„Plug-In instance“ - a currently active Plug-In. Analogous to a class instance.

Related Patterns

Plug-In Context: A single Plug-In is often not shippable on its own, but needs a (sometimes large) context of accompanying files.

Framework Application: The application is often implemented as Framework Application to enable a larger amount of reuse, and increase the time in market.

Pattern 2: Framework Application

Context

An application has factored out some functionality that is now implemented by Plug-Ins. Plug-Ins implement a specific functionality that requires usage of the application, like subclasses or specific parameterised instances of common application domain classes.

Problem

How can a Plug-In create and use application domain objects?

Forces

- The application knows and defines the domain
- time to market (for application)
- The Plug-In knows which domain objects it needs to employ, subclass, or instance
- The Plug-In must be as independent as possible from the Application, including internal implementation issues (may include even the operating system)

Solution

The application offers a framework. This is a black box framework offering no insights in the application, but defining opportunities for subclassing and parameterisation. Only part of the application is a framework. Other parts control loading and activating the Plug-Ins, or deal with completely unrelated stuff. Each interface for a Plug-In kind corresponds to a set of related „hot spots“.

Consequences

☺ Plug-Ins are easy to integrate with the application

☺ All Plug-Ins conform to the same interface

☺ Plug-Ins can be reused by other application that offer the same framework, allowing an option for Product Lines or Product Families

☺ Plug-Ins do not depend on application implementation - in extreme cases (OS hidden) allowing the application to be portable without the Plug-Ins even knowing about that

😐 Different Plug-Ins can hardly know each other, integration with cross-references requires special measures

☹ Development effort of application increases significantly, depending on the size of the framework part

Implementation

Reference books and articles on frameworks. (Pree, Johnson)

Parameterisation instead of subclassing: Is especially useful when Plug-In subclasses would have to use internal application services. Common example is persistence, which would require the Plug-In to change the database scheme. See section 3. (and reference to Component definition of Szyperski)

Example

OLYMP delivers a large number of application objects, that the Plug-Ins for specific local observation systems use. Some can be parameterised (like `Room`, `Alarm`, `Sensor`), others are intended for subclassing (like `PlugIn`, `CommunicationChannel`).

Known Uses

10 ProductL, ProductW

Related Patterns

See the factory and object creation patterns below

Pattern 3: Mutual Contract

Context

(Framework) Application and Plug-In projects are established, Plug-In purpose defined.

Problem

How does the application define the Plug-In interface?

Forces

- Stay independent of each others internals
- 20 Replace, remove or activate any Plug-In at run time
- Application uses and addresses the Plug-In
- Plug-In needs access to key classes and services of the application
- The application may need to be portable

Solution

Publish the interface the Plug-In is expected to fulfill, and the interface offered to it.

The Plug-In uses not only system services, but application services as well. Also the expected Plug-In functionality requires a custom interface. Figure 1 shows the major components, and their dependencies.

Plug-In Definition is the interface the framework requires from the Plug-In (Required Interface, 30 *UK*). The Plug-In is modeled as one or several abstract classes, together with their respective abstract factories or factory methods.

Plug-In adds specific knowledge to the application. It offers a factory or method that returns classes conforming to the expected interface. The internal implementation is hidden, and the visible class can serve as a Facade (*GJHV*, 185) to it. The Plug-In may use services of the application, and must use the domain objects the framework provides.

Framework Interface defines the services and domain objects of the framework (Offered Interface, *UK*). Their implementation is hidden from the Plug-In by abstract factories.

Implementation provides a process for execution, implements the framework services and domain objects, invokes the Framework Interface component, and activates the Plug-In by calling the factory and giving references to the framework objects.

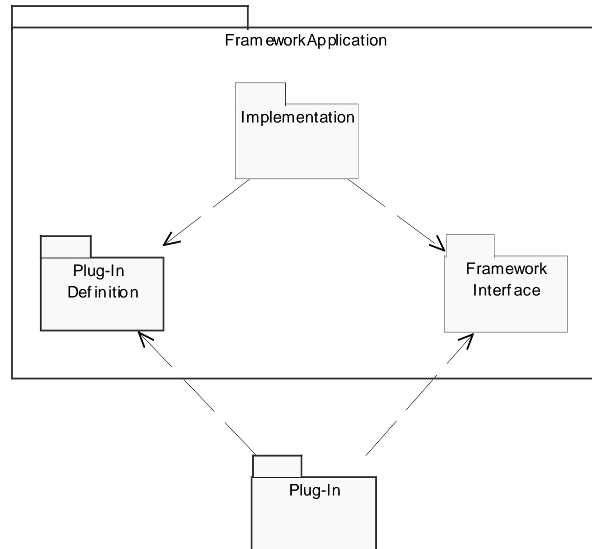

Figure 1: Major packages of a framework using a Plug-In.

All clients to the Plug-In can only access it through the Plug-In Definition component and interface, and the Plug-In can only access those instances and services published by the Framework Interface.

Are there foundations that the application is build upon, that can seriously be considered stable? If portability is not an issue, these are candidates to become part of the published Framework Interface. Class libraries are a perfect start to check.

Consequences

- ☺ Plug-In has access to application classes and services
- ☺ Clear dependency structure
- ☺ Internals of Application and Plug-In are invisible from the outside
- ☺ Plug-Ins can be added and removed at any time
- ☹ The application is bound to offered classes and services

Implementation

Keep track of the physical state of the Plug-In (loaded, active, inactive, unloaded) and make that become part of the Plug-In Definition. The Plug-In must have the opportunity to react on each transition. It is also useful for performance tuning (see Advent pattern, *KM*).

Example

Olymp offers base classes, and services like error log and alarm handler. It requires that Plug-Ins implement a subclass of LocalSystem, and overload virtual functions like check() and report().

Known Uses

ProductL, ProductW

Pattern 4: Essential Plug-In Functionality

Context

Application and Plug-In are in place.

Problem

Which is the user visible commonality of all Plug-Ins (of a kind)?

Forces

- All Plug-In types are different, and require different access
- Plug-Ins are developed independent of each other, by independent parties
- Steering different Plug-In types in the same direction is difficult, expensive, tedious
- Application and user need a common entry point

Solution

Define a number of expected functionality, classes, or objects, and make them a mandatory part of the Mutual Contract. Take whatever means to enforce this mandatory part technically.

Expected functionality from application side is the loading and activating procedure (see above).

Expected functionality from user side is a top-level configuration dialog at a defined navigation place. Depending on the application, a large number of additional dialogs and controls may also be expected common functionality.

Consequences

- ☺ Users can treat different Plug-Ins identical, enabling seamless integration
- ☺ Technical entry points are broadened and allow for convenient and easy to understand interfaces
- 😐 Application must have steering influence on Plug-In development
- ☹ A mutual contract can not be sufficient for a common look and feel. Additional style guides are required
- ☹ Only parts of this mandatory contract can be technically enforced

Example

All local observation systems that plug in the OLYMP central, need to provide a dialog with communication settings.

Known Uses

ProductL, ProductW, screen saver

Related Patterns

Loaded Factory: Defines the purpose of specific requested instances.

Mutual Contract: The expected functionality should become explicit part of the mutual contract. It is an implicit part of it anyway.

Pattern 5: Cooperating Plug-Ins

Alias: Plug-Ins Span Multiple Layers

Context

An application has defined Plug-In interfaces. Seamless integration requires specific additions beyond the Plug-In's model extension, like specific view and control, and possibly data exchange.

Problem

How can functional additions span multiple layers?

Forces

- The functional Plug-In interface should be concise and complete
- Specific data requires specific interpretation and specific view

Solution

Define a distinct Plug-In interface for each distinct area. Provide a common identifier so that the application can activate the appropriate counterpart.

This allows for extension specific data and classes added to the model. This data can only be added by an extension specific Plug-In, and be viewed by another extension-specific Plug-In. The application cares for the data exchange and processing in between, and ensures that the corresponding Plug-In gets in control on the viewing side. Each extension consists of one Plug-In type of each predefined Plug-In kind.

Configurable application domain objects need a reference to the extension identifier. The application must also ensure that distinct extensions come with distinct identifiers. The extension must ensure that no version conflicts between different Plug-Ins occur.

Avoid addressing all extension functionality through one interface - it would look like a swiss army knife. Separate into consistent domains (and employ further standard file formats, see Minestrone).

Consequences

- ☺ Each Plug-In interface is limited to one technical domain, and can be functionally closed
- ☺ Extension specific data can be passed through all system layers
- 😐 An extra level of packaging must be introduced
- ☹ Domain objects must identify to which extension they belong
- ☹ Integration between different Plug-In types of different extensions becomes impossible

Example

One of the local observation systems cares for the size of the rooms in which the smoke detectors are placed, and determines an event priority from this. To support this, the `Room` class must be subclassed, and this subclass must be fed and read by appropriate code knowing of this subclass.

Known Uses

ProductL, ProductW, Völter

Related Patterns

Minestrone: Defines the extension. Different parts of an extension are packed and shipped together.

Pattern 6: Plug-In Package

Alias: Suppengemüse[2], Minestrone, Plug-In Component

Context

Functionality is factored out, Plug-In interfaces are defined. Shipping a Plug-In as a stand alone extension component requires consideration of installation, localisation, …

Problem

10 *How to extend a Plug-In to turn it into a shippable component?*

Something is missing. Shipping requires installation. What about internationalisation? What about tiny little neat things like icons?

Forces

- Interface stability is critical to product lifetime
- Market demands short delivery cycles
- Custom interfaces require a learning curve
- Product acceptance begins with the installation procedure

Solution

Define and ship the functional extension as a package consisting of many files of many different 20 *types*. The central Plug-In is packed together with related executables, Plug-Ins, resource files, and „little helpers“. Application requests resources and „little helpers“ in standard formats.

To determine which files and file kinds to pack, start by identifying the functions throughout the life cycle, that the functional extension is expected to fulfil. Then try to find technical interfaces for these functions. Prefer technical standards of a long (expected) lifetime, and define custom Plug-In interfaces where necessary.

Typical aspects of life cycle support include:

- Installation program
- Plug-In (or a number of Cooperating Plug-Ins)
- Help text files (one per language)
- 30 Resource files (one per language)
- „Little Helpers“: Icons, sounds, movies, …

„Little helpers“: Whenever features are loosely coupled to the application domain, the application should avoid addressing them through customised Plug-In interfaces (and thus keep the custom interfaces minimal), but decide for a standard format to provide the feature. The Plug-In package must include the required files.

[2] Suppengemüse (English „Vegetables for a soup“), a bundle of celery, carrot, and leek. This German term reflects a consumers expectation: different things, bundled together for a common purpose.

Consequences

- ☺ Inherent cohesion of related Plug-Ins is maintained
- ☺ User experiences comfort and convenience
- ☺ Solution partly relies on existing standards, increasing the interface stability
- ☺ Application is open fur future use - the number of offered standard interfaces to future extension packages can be enhanced.
- ☺ Package parts can be developed in parallel, and by a wider range of developers
- 😐 The extension component becomes broad instead on complex, still requiring development effort and logistics
- ☹ Additional policy for versioning of the complete shipped package is required
- ☹ Parts of the interfaces are not controlled by the application

Example

When observed buildings are connected to the town central OLYMP, it is helpful when each audible and visible violation announcement allows for immediate recognition of the affected building. Thus, a specific sound, delivered in an additional WAV file, and an icon of the particular building, delivered in an additional ICO file, become part of the driver software for each building.

Known Uses

Like most complex applications, Microsoft Word consists of many different files and file kinds: Executables, help files, converters, dictionaries, registry entry file, document templates, and many more. An extension for a specific country also comes as a collection of files, like help file, menu file, dictionary file, hyphenation rules file, thesaurus file, and grammar file.

ProductL, ProductW

Related Patterns

Cooperating Plug-Ins: Minestrone is especially useful for them.

Section II: Organisational Issues

Pattern 1: Plug-Ins are Projects

Context

An application has defined interfaces for Plug-Ins. The market is demanding highly customised solutions

Problem

How can the application be adapted to fulfil single customers wishes?

Forces

- Customisation is expensive and costs time
- Code reuse can shorten this time

Solution

Implement customisation as Plug-In project. The complete application is the reused code, and the Plug-In is specifically developed for a particular customer, not to serve as a off-the-shelf product.

This is a greatest advantage of the Plug-In approach: A high amount of customisation becomes possible, where off-the-shelf application products can fulfil very specific needs for a reasonable price.

Consequences

☺ Off-the-shelf application is highly adaptive

☺ Customisation is highly cost effective

☹ Custom solution can not be reused for similar problems

Known Uses

Integration of networks and databases in technical domains, with corporative information systems.

Related Patterns

Yellow Code: can help to find even more options for code reuse

Plug-Ins are Products: more appropriate in other markets

Pattern 2: Plug-Ins are Products

Context

An application has defined interfaces for Plug-Ins. The market is demanding a high amount of available extensions.

Problem

How can a large number of Plug-Ins be developed and distributed?

Forces

- Developing Plug-Ins is expensive
- Shipping is expensive
- Applications are shipped in different versions
- Most customers demand similar extensions

Solution

Develop the Plug-Ins as sellable products. Identify the products with the highest potential, and ship the corresponding Plug-In as a separate product for users of the application.

Consequences

10 ☺ Software can be sold very effectively

☺ Market share of the application increases with Plug-In availability

☹ Stability and completeness of application interfaces is critical

Implementation

Determining the highest market potential varies vastly with the domain. It is different for the internet browser world (watch out for emerging file formats) than for the scientific laboratory (build relations to large analyser manufactures, to learn about the installed base and receive technical drafts of future products).

Known Uses

Screen saver. Popular also for video games: Flight Simulator, SimCity, Adventures can be
20 extended by plugging in additional environments („worlds“).

Pattern 3: Plug-In as Customer

Context

Framework and Plug-In build a mutual dependent system that only together are useful to the end user. Specific Plug-Ins are developed for a specific framework, and not usable in other contexts. Technically, the framework does not rely on a specific client Plug-In, and both sides are decoupled so that changes do not migrate. But both side's success is closely coupled.

Problem

How should the relations between the different development teams be organised?

Forces

30 Frameworks must live for long times, and for this be supported by a large number of Plug-Ins.

A Plug-In developer needs support to ensure his return on investment.

Solution

The framework supplier treats the development teams of Specific Plug-Ins as its customers. The framework delivers to the Plug-In developers, supplies technical and marketing support, and troubleshooting in a hot line manner.

The framework recognises that it does not deliver to end users, and tries to advertise and convince other development teams to start a joint effort.

Consequences

☺ Lifetime of the framework is increased

☺ Plug-In developer receive serious support decreasing their investment

Known Uses

Database vendors do parts of this job. Their products are not visible to end users, but their market share and success depends on the number and quality of applications developed with their engine. And most of them have managed to be highly visible, so that application developers can do advertising based on the „good name“ of the underlying database product.

Pattern 4: Template Code

10 #### Context

Employing a framework or application that is hard to understand. A successful framework needs lots of users to provide their specific services in the framework context.

Problem

How do you provide knowledge to your users?

Forces

- Frameworks must live for long times, and for this be supported by a large number of Plug-Ins.
- A Plug-In developer needs support to shorten his learning curve.
- Framework’s team time spend on support should be minimal.

20 #### Solution

Provide reuse on learning curve. Give sample code that can partly serve as production code, and that covers most technical and at least a few basic application areas of the framework.

This code is usually developed either way while testing the framework.

Consequences

Users receive a frame for development, which is far more that a framework for the application domain. Framework’s development time is used twice: both for own testing effort, and for customer support.

Known Uses

Very common

30 #### Related Patterns

Specific Plug-In: Serving developers of Plug-Ins.

Loaded Factory: The Template Code explains the loading and factoring mechanisms. Especially useful when factoring deviates from standards or common industry (like COM)

Pattern 5: Yellow Code

Context

Application and various Plug-Ins are developed and shipped. The number of new Plug-In development projects increases, and most of the different Plug-In types have some functionality in common.

Problem

How can you enable software reuse between isolated projects?

Forces

- Common functionality comes cheaper if code would be reused
- Reusable software would influence code ownership
- Different Plug-In projects are not coupled, and should not be

Solution

The application provides a collection of useful code from different Plug-Ins. Each Plug-In may decide to add its code to these „yellow pages“ of code, which would not cause the application to take the ownership of this code, but to include it in its pool of useful software. This pool is part of the „development kit“ for Plug-In development, and may be frequently updated. All newly developed Plug-Ins may use this reservoir.

Consequences

☺ Another support aspect of the application for new Plug-Ins, increasing the number of available Plug-Ins - and thus the application’s market success

☺ Decoupled projects have a way of gaining a mutual profit

☺ Code ownership is clear and remains stable

😐 The application is not extended by classes that do not belong to its „core business“

☹ Requires a cooperative culture of Plug-In developers

☹ Application has to spend effort in maintaining and distributing the reuse pool

Example

OLYMP offers some frrequently used classes for WAN communication to a local observation system.

Known Uses

ProductL, ProductW

Section III: Creation and Factories

Identifying the Plug-In, bringing it to life (activation), and creating all the objects that application and Plug-In exchange or use in common.

Pattern 1: Passive Registration

Also: Registry of Plug-Ins

Context

Application has defined Plug-In interfaces. Plug-Ins are available. User decides during run time which Plug-In to activate.

Problem

10 *How are the Plug-Ins known to the application?*

Forces

- User interaction becomes tedious for standard workflows
- Automatic installation requires development effort
- Startup time of the application and of a Plug-In should be minimal
- Application does not need to know about available Plug-Ins before activating one
- Plug-In registration does not demand information from the application

Solution

The application defines a place where it looks for available Plug-Ins. Each Plug-In installs itself there.

20 The simplest solution allows the application to scan a directory for a specific filename extension. For more convenient packaging, each Minestrone (Plug-In Package) may create a directory on its own.

Consequences

☺ Plug-In installation is very simple, and can be done with standard tools and batches

☺ User interaction is not required during installation

☺ Plug-Ins can be installed at any time

☺ Plug-In installation can be initiated remotely, enabling network computers

☺ Application startup time does not depend on available Plug-Ins

☹ Version conflicts may appear, requiring a resolution policy

30 **Known Uses**

Windows screensaver

Pattern 2: Active Registration

Alias: Feature Registration

Context

Application and Plug-Ins are available. Application decides due to external events which Plug-In it has to activate. Domain class instances are bound to a particular Plug-In type.

Problem

How are the Plug-Ins known to the application?

Forces

- Application decides due to external events, which Plug-In must be activated
- Plug-In availability must be known before activation (e.g. one of a set of Cooperating Plug-Ins)
- Automatic installation requires development effort
- Startup time of a Plug-In must be minimal

Solution

Plug-Ins register their existence, features and properties at the application during registration. The application provides interfaces that are available to installation programs.

Consequences

☺ Activation of a Plug-In requires no user interaction

☺ Activation time of Plug-Ins can be optimized

☺ Plug-Ins of correct type are certainly available when required

☹ Plug-In needs to be packaged with at least an installation program

☹ Plug-In installation requires the application (or some of its tools) to be active

Implementation

To minimise the activation time of Plug-Ins, see the Advent pattern (*KM*) to shift loads to more appropriate situations.

Example

The OLYMP town central knows in advance which local systems have to be connected when. This schedule can only be build when the Plug-Ins for the respective local security systems register themselves.

Known Uses

ProductL, ProductW

Related Patterns

Passive Registration: shows a much simpler way of registration, that is feasible if the requirements ever allow for it.

Mutual Contract: The application must include the installation services in the published interfaces.

Minestrone: Active Registration requires additional files to be shipped together with the Plug-In.

Pattern 3: Factories on Both Sides

Context

Plug-In and Application share a Mutual Contract.

Problem

How can the participants be isolated from each others implementation internals?

Forces

- Isolation is a major design goal, e.g. for portability reasons
- Performance at the Plug-In interface is not an issue
- Indirections and firewalls cause development effort

Solution

Both the application and the Plug-In access each others classes through factories.

Place factories for classes that are completely known within framework, on the framework side. This makes client code less dependent, and helps to hide implementation details (like the selected database product). Plug-In creates the objects by parameterisation of the Framework Application classes. This implies that the Plug-In decides about time of instantiation, though the application may indicate that the Plug-In should do so “now“.

Place factories for classes that only the Plug-In can know, on the Plug-In side. Framework Application decides when these are used. Factories on Plug-In side are usually “Pluggable Factories“ providing objects of a predefined purpose. Often the Plug-In factory is asked only for objects that the Plug-In previously has defined as “supported“ or “available“.

Consequences

☺ Application and Plug-In can define a Mutual Contract free of implementation details

☺ A high amount of independent development is possible

😐 Object creation costs slightly more performance

☹ Both sides spend implementation effort for firewalls

☹ The performance overhead becomes significant for small, very frequently created objects

Implementation

Typical example for the factory on Plug-In side is the `PlugIn` class itself, through which the application addresses all Plug-Ins uniformly. Also the essential features are created through factories.

Known Uses

ProductL, ProductW.

Related Patterns

Parameterized Object Creation: is the natural counterpart for the application side factory.

Loaded Factory: describes details of the factory on Plug-In side.

Pattern 4: Parameterized Object Creation

Context

Application and Plug-In in place and active. Plug-In needs specific instances of the predefined application domain classes. These domain classes are provided by the application via a factory.

Problem

How can the Plug-In easily create specific subclasses of application domain classes?

Forces

- Plug-In specific subclasses of base classes the application factors are difficult to implement
- Most domain objects can only be created with specific Plug-In knowledge

Solution

The application defines constructors where all relevant domain knowledge can be passed as arguments. The Plug-In creates domain objects with construction arguments.

Example: `Alarm`. Use this when the semantics are closed, and the Plug-In decides about the time of creation, i.e. its involvement is essential.

Consequences

☺ Creating specific instances is very convenient

☺ Factory mechanisms can be employed where necessary

☹ Subclassing is not supported

Variant

Objects need not be created by the Plug-In. The application can also create the domain objects based on properties the Plug-In provides. Use this when involvement of the Plug-In is minimal, and can be satisfied by a simple registration. The Plug-In does not even have to be active then. Most appropriate when multiple instances of the same Plug-In type can be present simultaneously. But not necessary - these objects can also be created at installation time.

Example: `Rack` and `LoadList` for analyser.

Known Uses

ProductL, ProductW. Both use the pattern, ProductL also the variant.

Related Patterns

Pattern 5: Subclasses Through Aggregation

Context

Application and Plug-In in place and active. Plug-In needs specific subclasses of the predefined application domain classes. These domain classes are provided by the application via a factory.

Problem

How can the Plug-In create specific subclasses?

Forces

- Plug-In specific subclasses of base classes the application factors are difficult to implement
- Plug-In specific subclasses can not use internal application services, like persistence
- Predefined application classes are hardly sufficient for a complete domain model

Solution

The Framework Application provides attachable classes, that each domain class instance can aggregate. The Plug-In defines subclasses and implements them in terms of a parameterized base class with specifically aggregated attachable class instances.

10 Variant: Framework Application adds additional members to the domain classes, that have no defined semantic meaning. Each Plug-In may use them for individual interpretation.

Both approaches only work with Cooperating Plug-Ins, so ensure that application visible data is created, changed, and presented with identical semantic interpretation. Integration of data from different Plug-Ins is limited to the semantics known to the application.

Consequences

☺ Subclassing becomes possible without breaking the applications integrity

☹ Development on Plug-In side is high

☹ Integrating objects from different Plug-In types becomes hard, depending on the amount and depth of subclasses

20 **Implementation**

Resist the temptation to broaden the Framework Application interface so that a Plug-In may define own derived classes. Considering persistence, such a broad interface would include access to DDL ("create table" command). This can cause serious side effects between different Plug-In types, and takes away all tuning options from the application. Further development and improvement of the application is extremely difficult with broad interfaces.

In some respect, this would be trying to get across the limitation that components do not have persistence (Szyperski). Plug-Ins have commonalities with components here.

Example

A local observation system may want to add the property `volume` to the domain class `Room`, to
30 double check for the created alarm's priority.

Known Uses

ProductW

Related Patterns

Essential Interfaces. Factories on Both Sides

Pattern 6: Loaded Factory

Context

An application deploys Plug-Ins. These must be added or removed at runtime.

Problem

How can an application load and unload plugged-in components? How can the Plug-In provide specifically the objects the application needs to employ?

Forces

- The application requires the Specific Plug-In component to provide specified services.
- Implementation of services is entirely up to the added component.
- Standard component activation techniques are not available or insufficient
- Needs to deliver instanced objects to abstract classes. Parameters passed to identify the instance.

10 #### Solution

Ask the Specific Plug-In for an instance of a predefined Abstract Factory. Use operating system or environment facilities to load the Specific Plug-In into memory, ask it for the factory, and continue by using that factory instance to provide specific instances. Extend the Abstract Factory definitions to provide both subclass instances, and objects for a particular purpose.

This solution ensures that the contract is fulfilled by the plugged component with respect to the class interface, while the implicit extensions to the Mutual Contract (the Essential Interfaces) can be expressed in terms of a defined purpose for the requested instance.

Besides DLL or RTL loading (under NT and Solaris respectively), also active objects could be employed. This requires an additional translation layer on the Framework side, because each
20 instance requested from the factory matches to a component definition, so that each call to a member function must be translated. In these cases it is more efficient to resize the Plug-In interface, and ask for specific instances directly. The original Plug-In becomes a Plug-In collection then, which can be packaged in terms of Minestrone.

Consequences

Implementation

Variants: runtime (completely pluggable) and startup time (previously linked in).

Runtime variant includes somehow getting the loading signal and a name. This name is used to identify the (DLL, whatever) and ask it for its factory. Startup variant scans all linked
30 components during initialisation, and asks them for their factories.

Variants are identical afterwards. The framework decides which objects are needed, and requests them from the loaded factory. The loaded factory may decide whether it creates an instance of a specialised subclass, or whether a parametrised class instance can do the job.

Special: the loaded Plug-In (or in larger context, the installation program of the Minestrone) may announce special Plugged-In objects beyond the instances the framework requires as mandatory. These are then also delivered by the loaded factory.

Known Uses

Especially in embedded areas where expensive off-the-shelf mechanisms can not be used, because of price or environment.

Conclusion

What have we gained? - A technique to separate custom parts from applications, that allows as well for a high amount of customization as for future extensions.

What is still missing? - Depending on the answer, this could be the start of a pattern language for application framework development. Some things will be adaptable from [BEP], others are specific here.

Relation Plug-In to highly reusable applications: For very long, applications have been used in larger context. Development shells employ editor, compiler, and linker, their major value being to configure which applications to call. The applications are perfectly stand-alone (I like to call them Reusable Application), and are not Plug-Ins (that can live only in the context of their application).

Relation Framework Application to framework: The difference between application and framework with respect to Plug-Ins is diminishing. A framework defines the abstract classes and the collaboration structure (Ralph Johnson). The Framework Application does just that, but then adds the flow of execution, i.e. processes, and tasks. Applications provide major functionality on their own, where frameworks need Plug-Ins to be of any use. Frameworks come to „life“ when a semantic Application employs them; Plug-Ins come to life when the application activates them.

Relation Plug-In to Component Based Development: There is a focus on Component Based Development. Plug-Ins are similar in a lot of respects. They are pluggable, cover arbitrary functionality, and allow for larger amounts of code reuse than other, purely OO based techniques. Other aspects are different. Pluggability is limited to a specific Framework Application. They are not freely pluggable, but limited to their domain and application. While this approach is narrower than CBD, the range of applicability is broader. Plug-Ins even appear in embedded systems, where today’s Components are mostly limited to distributed enterprise models. Nevertheless, the cross section is visible. When Component Based Development further succeeds, the mechanisms available for Plug-Ins will become more convenient, and the border between a Component Based Application and a Plug-In Based Application will be very thin. Main difference is then in the proprietarity.

Where do we go from here? - One future task is a pattern collection for Product Lines. Some preconditions are given here, but a Framework Application and a number of Plug-Ins do not make a product line. A successful product line implies reusable Framework Application while keeping the interface to Plug-Ins stable.

Acknowledgements

I would like to thank my current and former colleagues for the successful and exciting work. These pattern are late results of our shared experience. Special thanks to Jens Coldewey for introducing patterns to me, to Neil Harrison whose careful shepherding lead the sheep safely towards the EuroPLoP conference, and to John Vlissides for.

References

GHJV Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison-Wesley 1994

JL John Lakos: Designing Large Scale C++ Applications, Addison-Wesley 1996

KM Klaus Marquardt: Patterns for Software Packaging, Installation, and Activation. In: Proceedings of EuroPLoP 1998

NW James Noble, Charles Weir: Proceedings of the Memory Preservation Society. In: Proceedings of EuroPLoP 1998

UK Ullrich Köthe: Design Patterns for Independent Building Blocks. In: Proceedings of EuroPLoP 1998

BEP Kyle Brown, Philip Eskelin, Nat Pryce: Component Design Patterns. Work under construction in Wiki Wiki Web (http://c2.com/wiki?ComponentDesignPatterns)