Patterns for Plug-Ins

Klaus Marquardt
Email: marquardt@acm.org

Abstract

This pattern collection helps to define, implement and package Plug-Ins specific to an extensible application. Central patterns are the Plug-In and the Plug-In Contract between the Plug-In and the application; afterwards patterns for packaging and registration of Plug-Ins are explored. Process and organisation patterns complement the general technical patterns, and support the technical flexibility introduced with Plug-Ins. The patterns in the last chapter focus on implementation techniques and shows how other design patterns can be used for Plug-Ins.

Introduction

Software is cheap. Ever more functions in technical devices are implemented in software, because software is much more flexible than hardware or mechanics. But this flexibility, adaptability and extensibility does not come for free. The costs in software development and ownership along a product's lifetime are significant and often underestimated. Measures to minimise them are of high interest in the industry.

Flexibility in software can be achieved by careful design, by configurability, and by using software components. These techniques supplement each other, and a careful design is prerequisite for the latter ones.

Carefully designed software separates different aspects explicitly and anticipates certain kinds of changes. Because other changes require rewriting parts of the software, management of internal dependencies is a key to maintainability. Each change migrates through the complete line from development to the customer.

Configurable software determines parts of its behaviour at run time when it reads and interprets its configuration. The initial development and test effort is high, but a larger variety of changes can be treated without changing the delivered software itself.

Using software components requires dividing the software into parts that can be developed and exchanged independently. The gained flexibility is virtually infinite as components contain executable code. Preparing for exchangeable components increases the development effort, but the ability to change the software by adding or exchanging distinct parts of it reduces the costs during the product lifetime.

The power of the Plug-In component approach is amazing. Software systems can be delivered almost "nakedly" and most user value is added by Plug-Ins that are developed separately. Existing applications can be extended without change to support new file formats, new customer devices, or new processing abilities.

The Patterns for Plug-Ins treat the role of software components in the context of a hosting software, ranging from the definition to shipment and organisational issues.

Roadmap

The pattern collection is divided into several chapters with specific topics:

Chapter I, “General Plug-In Techniques”, deals with the definition of a Plug-In and its context, relations to the application, packaging and activation. The patterns in this chapter are of a general technical nature, applicable in a wide range of development environments and methods.

Chapter II, “Organisation and Process”, shows how development can be managed by different subprojects, and gives hints how the project can interact successfully.

Chapter III gives concrete implementation patterns, and presents techniques that also employ other design patterns useful for Plug-Ins.

Example

The patterns usage is illustrated by the fictitious ARGUS example. ARGUS is a security central that integrates security systems of single buildings. Within each building an independent local system observes doors, windows, and parameters like temperature and humidity to identify (possibly illegal) access, fire or other problems. The ARGUS central connects to a number of individual local security systems, retrieves their data, and reports all violations both visibly and audibly, depending on their priority.

The local building security systems come from different vendors and have different sizes and abilities. At the ARGUS central each of them is represented by a Plug-In component, that cares for the communication to its corresponding local system and for the translation of received data into the model of the ARGUS central.

The code examples are written in C++ for a Windows platform and make use of dynamic link library (DLL) technology.

Known Uses

The use of Plug-Ins is common through a large variety of applications. The patterns refer to the following known uses:

Adobe Photoshop provides functions to manipulate photographs.

Word and *Rational Rose* like other applications provide extensibility through an application specific interpreted language (Basic) and customisable GUI elements (menus and tool bars).

Netscape and other browsers present web pages that combine text, pictures and movies with navigation facilities and elements that initiate an action.

Windows, pSOS, TOS, Unix … virtually every operating system can be adapted to different hardware devices and allows the user to start programs.

*OpenCard*s [*OCF98*] that are physically plugged into a defined interface carry data and code that an application can read or execute.

LabPlug[1], a laboratory automation system, controls a number of chemical analysers, and integrates them into a laboratory or hospital information system (LIS, HIS).

MedPlug[2] is a family of medical devices that observe and control a patient's health. Each MedPlug device controls a (possibly changing) number of replaceable sensors and actors or packages of them, and integrates them into an intelligent user interface.

n-sell is an e-commerce system that is customisable for different businesses [*Völter99* also provides an extensive source code example].

[1] Name changed to protect the non-disclosed.
[2] Name changed to protect the non-disclosed.

Chapter I: General Plug-In Techniques

Figure 1 shows the patterns in this chapter, and their main relations. Plug-In describes how functionality can be added to applications at run time. Each Plug-In lives in a context, the activating application. This Framework-Providing Application provides access to services and domain objects. Framework Application and Plug-In share a Plug-In Contract defined by the application, that describes duties that each Plug-In has to fulfil, options for specific extensions, and functions and libraries that the application offers to all connected Plug-Ins.

The Plug-In Registration enables the application to find its functional extensions. The registration starts the Plug-In Lifecycle, which is part of the Plug-In Contract.

A single Plug-In may not be sufficient to fulfil the complete extensibility task. Large functional parts can be separated into multiple cooperating Plug-Ins, with One Plug-In per Task. These Plug-Ins are accompanied by additional programs and files forming a shippable Plug-In Package.

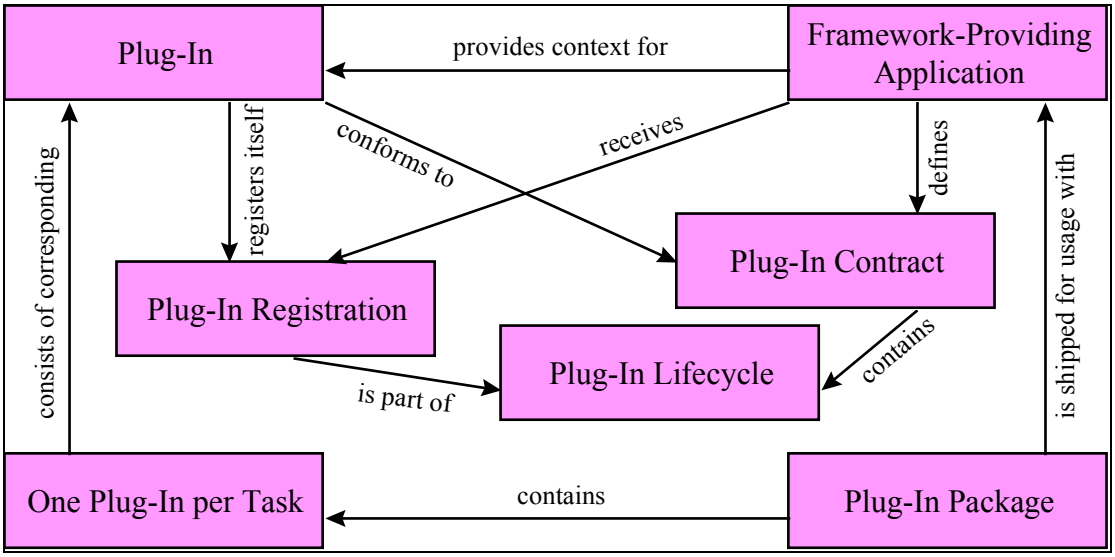

Figure 1: Roadmap for General Plug-In Techniques

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 functional component is available
- Early delivery increases market share and profit
- Specifically added functionality can not be foreseen
- Which functionality is dynamically added when, is determined at run time and can hardly be foreseen
- Kind of additional or exchanged functionality is well known
- A technology evolves, the application will be used in unforeseen ways
- Shipping is expensive
- The application is not changed 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 (Plug-In Contract).

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 not depend on a Plug-In internals, and often not on the presence of a particular Plug-In kind. Plug-Ins can be used to factor out essential functionality. In this case the presence of a particular Plug-In is required, and the application is always shipped with that Plug-In.

Terminology „Plug-In kind“ - different Plug-Ins are of the same kind when they conform to the same predefined interface. The OO analogon would be a superclass.

„Plug-In type“ - the Plug-In implementation denotes the type. The OO analogon would be a derived class.

„Plug-In instance“ - a currently active Plug-In. The OO analogon would be a class instance.

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
- 😐 Delayed developed Plug-Ins 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

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.

Plug-In can be implemented using any OO or component technology, such as DLL or run time library, or active objects (e.g. Active-X). The application decides about the activation time and conditions (Plug-In Lifecycle). A Plug-In may start and use helper applications when useful.

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

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, occasionally a resource budget is defined as part of the interface. The application can also define the outline of the logical design, but internals are completely up to the Plug-In.

Variants: 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.

Organizational Issues

Development of customer specific solutions: Development of Plug-Ins can be organised as separate projects, after the application is available. Off-the-shelf Applications can then be adapted to a specific customers needs (Customisation through Plug-In), enabling a tremendous amount of reuse, the whole application, and minimal development time and effort.

Plug-Ins can serve as separate products, that either the application vendor or an independent manufacturer can sell. The first case is common with video games and in narrow domains, the latter in often used technical domains (like screen savers and web browser Plug-Ins).

When applications are shipped together with available Plug-Ins, Plug-Ins are developed as sub-projects simultaneously to the application project (developing the market visible product). Especially in non-technical domains, the applications market success can depend heavily on the existence and number of available Plug-Ins. In this case the application project has to take care of Plug-In projects as preferred customers.

Known Uses

Adobe Photoshop uses Plug-Ins extensively to factor out internal as well as extensible functionality. Development of Plug-Ins is considered part of the application project. Some external developed Plug-Ins can be purchased, e.g. Kai's Power Tools.

Netscape places viewing functionality into Plug-Ins, each interpreting a specific graphics or movie format. Third parties provide additional Plug-Ins for less common or new formats.

Device drivers for most operating systems are Plug-Ins provided by the hardware vendor. Each commercial application (like Word) is a Plug-In to an operating system.

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.

The OpenCard standard defines the Plug-In interface that the Plug-Ins, code on the OpenCard that is physically plugged into the system, comply with. Activation of the Plug-In is explicitly done on applications request.

LabPlug 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.

MedPlug 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.

n-sell is extensible to exchange ordering and billing data with different host systems like SAP.

Example

The ARGUS central can connect to a variety of different observation system. The specific, mostly proprietary transmission protocols are factored into Plug-Ins. This way the ARGUS 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).

ARGUS defines a superclass `LocalSystemPlugin` for this Plug-In kind from which local systems must derive:

```
class LocalSystemPlugin {
public:
    virtual void initialize( const Services &) = 0; // pass ref to services
// ...
    virtual ~LocalSystemPlugin() = 0;
};
```

The Plug-In representing the local system is placed in a DLL that exports a trader function. This function must have the same name for all Plug-In types, and it returns a Plug-In instance of the specific type. The passed specification defines the name of the local system, and communication settings like the network address.

```
#include "PluginDefinition\LocalSystemPlugin.h"
_declspec(dllexport) LocalSystemPlugin* getLocalSystem( const Specification &);
```

For the application implementation convenience, the Plug-In class and the DLL loading is encapsulated by a class `LocalSystem` that forwards all requests to the Plug-In and has additional member functions for loading and unloading. The DLL name is passed with the Specification, and `GetProcAddress` obtains the published trader function.

Related Patterns

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

Plug-In Contract: The interface that the Plug-In must conform to, and the interface the Plug-In may use to perform its task, are defined by the application.

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

One Plug-In per Task: To keep each Plug-In interface and duties compact and concise, an extension that covers multiple functional layers can be split into a number of corresponding Plug-Ins.

Pattern 2: Plug-In Contract

Context Application and Plug-In projects are established, Plug-In purpose defined.

Problem *How does the application define the Plug-In interface?*

Forces

- Different Plug-Ins are developed by different teams or companies
- Development must be decoupled in order to develop faster
- Development must be decoupled in order to keep the system manageable
- Steering different Plug-Ins in identical direction is difficult, expensive, and tedious
- Replace, remove or activate any Plug-In at run time
- Users expect common behaviour from different Plug-Ins
- Application and users need a common entry point for each 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.*

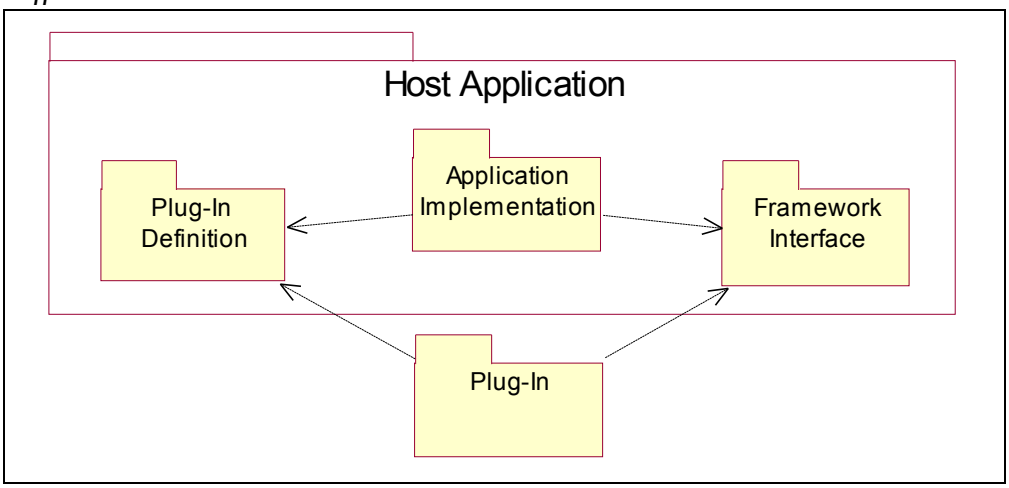

Figure 2: Major packages of a Host Application using a Plug-In.

The Plug-In uses not only system services, but application services as well. Also the expected Plug-In functionality requires a custom interface. Figure 2 shows the major components, and their dependencies.

Plug-In Definition is the interface the framework requires from the Plug-In (Required Interface, [*Köthe98*]. 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 [*Gamma+94* p185] 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. Their implementation is hidden from the Plug-In by (abstract) factories [*Gamma+94* p87, 107] or product traders [*Bäumer+97*].

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.

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.

Besides purely technical interfaces, the application defines an expected user visible functionality that each Plug-In has to provide. Common example is a top-level configuration dialog at a predictable location in the applications menu structure. Depending on the application, a large number of additional dialogs and controls may also be expected essential functionality. Take whatever means to enforce large parts of the contract technically, for example by defining required products from the Plug-In (see [*Bäumer+97*]).

Consequences

- ☺ Clear dependency structure
- ☺ Internals of Application and Plug-In are invisible from the outside
- ☺ Plug-Ins can be added and removed at any time
- ☺ Users can treat different Plug-Ins identically, enabling seamless integration
- ☺ Plug-In has access to application classes and services

- ☹ The application is bound to the classes and services it offers, its evolution potential is limited when the interfaces are published
- ☹ Only parts of the contract can be enforced technically, application should have some organisatorial influence on Plug-Ins
- ☹ A Plug-In Contract is not sufficient for seamless integration and common „look and feel“. Additional style guides are needed

Implementation

Keep track of the state of the Plug-In (Plug-In Lifecycle) and make that become part of the Plug-In Definition. The application calls these transitions to control the Plug-Ins.

A stable Plug-In Contract is critical to a Framework-Providing Application’s lifetime. This lifetime can be increased by simple interfaces with flexible parameters.

Example

ARGUS offers superclasses, and services like error log and alarm handler. It requires that Plug-Ins implement a subclass of `LocalSystemPlugin`, and overload polymorphic functions that the application calls.

Here’s the `LocalSystemPlugin` class in more detail. It allows the application to control the way of communication and to initiate functions that are executed asynchronously.

```
class LocalSystemPlugin {
public:
    enum State {
        undefined,  // before initialization
        inactive,   // no connection
        online,     // reports on communication events
        offline     // reports on application requests only
    };
// ...
    virtual State setState( State) = 0;
    virtual State getState() = 0;
    virtual void initiateStatusReport() = 0; // demand a system status report
    virtual void initiateSensorStatus( Sensor &) = 0; // demand sensor report
    virtual inhibitLocalMonitor( bool) = 0; // switch local monitor on/off
// ...
};
```

The application must in turn provide references to interfaces that the Plug-In can use to fulfil the requested tasks. These references are provided by the `Services` class.

```
class Services {
public:
    const ErrorLog& getErrorLog();
    const AlarmHandler& getAlarmHandler();
// ...
};
```

Known Uses

The contract of an operating system for executables includes the load procedure, where the system expects code and data segments, and the API the system offers.

LabPlug and MedPlug both offer access to their application objects, to services like an error log, and a library of customised GUI widgets. In turn, they expect Plug-Ins to create application object instances from the received data, and require usage of the custom widgets by convention (style guide).

Related Patterns

Plug-In: Plug-In Contract defines the boundaries for the Plug-In.

Framework-Providing Application: Plug-In Contract defines its services and dependencies towards the Plug-In.

Plug-In Lifecycle: is an essential part of the Plug-In Contract.

Pattern 3: Framework-Providing Application

Alias: Host Application, Plug-In Context

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 the 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 host 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“ [*Pree97*], [*Roberts+97*].

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 do not know each other. Applications integrating them with intelligent combinations need to take special measures
- ☹ Development effort of application increases significantly, depending on the size of the framework part

Implementation

Check out the standard literature on framework development (like [*Pree97*], [*Johnson99*]).

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 below, and [*Bäumer+97*], [*Szyperski98*], [*Szyperski99*]).

Variant **Safe Framework-Providing Application**

When the application has no control over future Plug-Ins, but will be held responsible for their failures, it needs to protect itself against „bad“ Plug-Ins. This is done by adding Facades [*Gamma+94* p185] that narrow and control the access from Plug-In to the application. The key advantage of the Safe Framework Application is that the Framework Interface (see Plug-In Contract) can be published. Disadvantages are increased development effort, performance penalties, and possibly lessened functionality that the Plug-In is allowed to provide or use.

Example

ARGUS 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 `LocalSystemPlugIn`).

The framework defines an `Alarm` class that is constructed via a factory (to hide persistence details):

```
class Alarm {
// ...
    const bool isVisible();
    const bool isAudible();
    void confirm(); // user has seen and confirmed this alarm
    void remove(); // creator/owner: alarm cause has vanished
};

class AlarmFactory {
// ...
    Alarm& createAlarm( LocalSystemPlugin& owner, int resourceId, int
priority);
// ...
};
```

The specific Plug-In creates its `Alarm`s during initialisation or registration (Plug-In Registration). Status reports and communication messages are converted to `Alarm` function calls.

Known Uses

All of the known uses are host applications. The amount of offered context respectively framework varies.

Operating system define no application but provide a technical context.

Applications like Word and Netscape define an application domain.

LabPlug and MedPlug provide a framework. LabPlug implements the “safe” variant: access to most objects is limited to reading attributes and changing only that subset of them where a connected analyser is the only valid source of information.

Related Patterns

Plug-In: is hosted and employed by the Framework-Providing Application.

Plug-In Contract: defines the relation between Plug-In and Framework-Providing Application.

Plug-In Registration is used to make Plug-In known to the Framework-Providing Application.

See also the specific implementation patterns in chapter III.

Pattern 4: Plug-In Registration

Alias: Registry of Plug-Ins

Context

Application has defined Framework Interfaces and Plug-In Definitions. Plug-Ins are available. User or application decides at run time which Plug-In to activate.

Problem

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 using 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.

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

Implementation

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

Variant

Active Registration

When the application decides due to external events which Plug-In must be activated, it can be important that the Plug-In availability is known before activation, especially if it is one of the cooperating Plug-Ins within a Plug-In Package.

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

To minimise the activation time of Plug-Ins, combine the Plug-In Lifecycle and Advent [*Marquardt98*] patterns to shift loads to the most appropriate situations.

This variant has its own 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
- ☺ Version conflicts can be resolved early, during installation
- ☹ 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

Example

The ARGUS 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. Thus, the Active Registration variant is chosen.

Known Uses

Operating systems have defined locations where device drivers have to be present. Executables are also registered passively by placing them into the directory structure where the user can find and activate them.

Atari TOS checks at boot time for presence of accessory applications in a certain directory.

OpenCard uses passive registration when physically connected. Activation of the Plug-In is explicitly done on applications request.

LabPlug and MedPlug need to change configuration data of the Framework-Providing Application and use active registration.

Related Patterns

Plug-In Contract: The application must include the installation services in the published interfaces.

Plug-In Package: Active Registration variant requires additional files (like the registration program) to be shipped together with the Plug-In

Pattern 5: Plug-In Lifecycle

Context Plug-In Contract is defined. Application needs to make use of Plug-Ins.

Problem *How can the application invoke and control the Plug-In?*

Forces

- Plug-Ins can be installed, activated, and deactivated during application runtime
- Plug-Ins need to take special actions in different employment phases
- Application needs to retrieve installed Plug-Ins and activate them
- Application needs to check and control a particular Plug-In types state as well as a Plug-In instances state.

Solution *The application defines the life cycle of the Plug-In*. The life cycle for a Plug-In instance contains loading, activation, deactivation, and unloading. The life cycle for a Plug-In type includes registration when Active Registration is chosen. The transitions correspond to member functions within the Plug-In Definition to allow the Plug-In to react.

Both cycles must be cleanly differed especially when registration is done in the Active variant manner and may occur during run time of the application. They may be merged when only one Plug-In instance per kind may be active at a time.

Consequences

- ☺ The Plug-In lifecycle is defined and controllable
- ☺ The application can control the Plug-Ins state, and the Plug-In can react on it
- ☺ The application can check the states of all registered Plug-Ins

Implementation The application checks at startup what Plug-Ins are registered, and offers them to the user. The user selects one or more to become active, do its job, and become inactive again. The application invokes the Plug-In without knowing more than the registered information that is also displayed to the user. The Plug-In decides itself about its normal inactivation.

The Plug-In can take advantage from the opportunity to react on each transition with respect to performance tuning [*Marquardt98*, Advent]. Time consuming tasks can be preferred or deferred. The applications loading policy is important here, so that a published “performance style guide” becomes useful [*Noble+98*].

Variant **Scheduled, Automatic, Event Driven**

Scheduled variant: After the user has selected the desired Plug-In, activation is delayed according to a schedule, or until an external event occurs.

Automatic variant: The application detects during normal operation (data procession) that it needs to activate a specific Plug-In. The appropriate Plug-In is determined from registration data.

Event Driven variant: Applications receive external requests (like a network event) to search for registration of a specific Plug-In, and starts it. The Plug-In does its job until the application receives a request to stop it, or an error condition occurs.

Example

ARGUS connects to local systems according to a schedule. The user configures at which time which registered Plug-In should become active. For performance issues, an additional communication state is introduced that controls the local systems responsiveness.

```
class LocalSystem {
public:
    enum State { unloaded, loaded, active };
    enum CommunicationState {
        undefined,  // before initialization
        inactive,   // no connection
        online,     // reports on communication events
        offline     // reports on application requests only
    };
// ...
    virtual void setState( State) = 0;
    virtual CommunicationState setComState( CommunicationState) = 0;
    virtual CommunicationState getComState() = 0;
// ...
};
```

Known Uses

Adobe Photoshop, Word, Rational Rose, OpenCart, operating systems and LabPlug activate Plug-Ins on users demand.

Screensavers are invoked according to a timer, i.e. scheduled.

Browsers automatically activate their Plug-Ins when the corresponding page contents appears.

MedPlug activates a Plug-In when the sensor presence has been indicated by a communication event, and deactivates it when the communication has ended.

Related Patterns

Plug-In Contract: Plug-In Lifecycle is an essential part of the contract.

Plug-In Registration: is the first step in the lifecycle of a Plug-In type.

Pattern 6: Plug-In Package

Alias: Plug-In Component

Context

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

Problem

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

- Strive for stable interfaces to increase the application's lifetime
- Customised interfaces may lack stability when standards are available
- Custom interfaces require a learning curve of the Plug-In developer
- Separate interface parts allow parallel development of different Plug-In parts
- End product acceptance is increased by comfortable usage

Solution

Define and ship the functional extension as a package consisting of many files of many different types. The Plug-In interface consists of the custom Plug-In Definition classes, and a number of additional files. 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 use custom Plug-In Definitions 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)
- 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 Definitions (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.

Consequences

- ☺ 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
- ☺ Inherent cohesion of related Plug-Ins is maintained
- ☺ User experiences comfort and convenience
- 😐 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

Implementation

While developing Plug-Ins, take care to constantly integrate the whole Plug-In Package and keep it up to date. The Plug-In Package it the granule of release.

Example

When observed buildings are connected to the town central ARGUS, 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.

A Plug-In Package for Rational Rose could combine a Basic program script and an installation program extending the user visible menus.

LabPlug requires one Plug-In for communication to the analyzer, one for specific GUI screens, an icon used by the “lab overview” screen, and an

installation program that adds the analyzer's properties to the common database.

Plug-In Packages for MedPlug consist of several Plug-Ins, some of them optional, and resource files containing language specific texts for the alarms that the connected sensor package may issue.

Related Patterns

Plug-In: One or multiple Plug-Ins are the central part of the package.

One Plug-In per Task: Plug-In Package is especially useful for packaging related Plug-Ins together.

Pattern 7: One Plug-In per Task

Alias: Cooperating Plug-Ins, Plug-Ins Span Multiple Layers

Context

An application has defined a Plug-In Definition. 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 Definition should be concise and complete
- Specific data requires specific interpretation and specific view
- Swiss army knife interfaces are difficult to learn and handle

Solution

Define a distinct Plug-In Definition for each distinct task or domain. 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 Plug-In Package).

Consequences

- ☺ Each Plug-In Definition 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

Implementation

The application must define the division into several Plug-Ins. Each Plug-In kind gets its own Plug-In Contract. For development and learning

efficiency, only the Plug-In Definitions (see Plug-In Contract) deviate between the Plug-In kinds, whereas the Framework Interface is the same for all of them. Dividing the Framework Interface into different sections, and documenting which section is useful for which Plug-In kind, further flattens the learning curve.

On the Plug-In side, typically all cooperating Plug-Ins are developed by one team, and share significant amounts of code. This pattern gives each single Plug-In a distinct technical domain focus, and helps to separate different concerns.

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, read and displayed by appropriate code knowing of this subclass.

For this kind of extensibility, the ARGUS system prepares by separating each functional extension into a communication Plug-In, an application Plug-In, and a GUI Plug-In. All these Plug-Ins work together, the functional extension is opaque to other system components. This way functional specifics of a plugged addition can be handled in all functional layers.

Known Uses

LabPlug and MedPlug define distinct Plug-Ins for communication and for display purposes.

Related Patterns

Framework-Providing Application defines the division into multiple cooperating Plug-Ins.

Plug-In Package ensures that different parts of the extension are packed and shipped together.

Chapter II: Organisation And Process

Customisation through Plug-In as well as Sell Plug-Ins and Plug-In as Customer explore the organisational aspects of the general Plug-In pattern in more detail.

Template Code is a common approach for customer support of libraries and frameworks. The idea is to provide some code that shows important principles and can easily be changed and extended by developers. Non-Profit Code Library is a more sophisticated approach where a secondary class library is build from already developed Plug-Ins, and code ownership changes towards the Framework-Providing Application that maintains the additional classes in a way similar to yellow pages – optional but useful for most clients.

Pattern 8: Customisation through Plug-In

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

Non-Profit Code Library can help to find even more options for code reuse

Sell Plug-Ins may be more appropriate in other markets

Pattern 9: Sell Plug-Ins

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.

Selling Plug-Ins is often no option for the application vendor itself. Each application must be provided with a number of the most important Plug-Ins as integral part of the product. But a wide spread application may create new markets for Plug-In vendors.

Consequences

- ☺ Software can be sold very effectively
- ☺ Market share of the application increases with Plug-In availability
- 😐 Application vendor and Plug-In seller may differ
- ☹ Stability and completeness of application interfaces is critical
- ☹ Few markets allow to get money for Plug-Ins

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).

Be prepared that the profit may better be expressed in terms of visibility and marketing than in money.

Known Uses

Screen saver. Browsers. Some video games sell additional levels, or worlds, as a separate product.

Some Plug-Ins for Adobe Photoshop can be purchased separately, e.g. Kai’s Power Tools.

Pattern 10: 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

- 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 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. During development, a subscription model to distribute intermediate versions with added functionality or changed interfaces is used.

Consequences

☺ Lifetime of the framework application is increased

☺ Plug-In developer receive serious support decreasing their investment

Known Uses Adobe. LabPlug. MedPlug.

Pattern 11: Template Code

Alias: Boilerplate Code

Context

Employing a framework or application that is hard to understand.

Problem

How do you Plug-In developers know how to employ the application?

Forces

- Applications must live for long times, and for this be supported by a large number of Plug-Ins.
- A (Plug-In) developer employing the application needs support to shorten his learning curve.
- Application's team time spend on support should be minimal.

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
- ☺ Learning curve of users is increased
- ☺ Application development test effort and code can later become customer support

Known Uses

Very common for all kinds of class libraries and frameworks

Pattern 12: Non-Profit Code Library

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 among Plug-In developers
- ☹ Application has to spend effort in maintaining and distributing the reuse pool

Example

ARGUS offers a class library for WAN communication to a local observation system. Plug-Ins for different local systems use these classes for their implementation.

Known Uses LabPlug, MedPlug.

Rational maintains a web site where users of Rational Rose can upload and download scripts and exchange tips and tricks.

Chapter III: Patterns for Plug-In Implementation

After introducing the Traders on Both Sides pattern, this chapter focuses on object creation patterns that guide application developers in defining the Plug-In Contract..

The application is the owner of the architecture, of technical solutions and services. It should only publish its essential technical abilities to Plug-In developers as part of the contract, and hide most aspects to keep the contract small and stable. A focus on the big picture – the covered application domain – leads to encapsulation of most of the infrastructure, often beginning with persistence mechanisms, going to class libraries and sometimes also including operating system.

The Plug-In is the owner of the application know-how. It adds the semantic knowledge using the predefined application domain model explicitly, while relying on the technical infrastructure only implicitly. Ideally, a Plug-In developer does not need to see any technical details of the application.

Problems with the object creation occur whenever the semantic knowledge of the application and the Plug-In do not match their technical abilities. Plug-In Defines Subclasses and Plug-In Parameterizes Application Classes address this problem in different lights.

Pattern 13: Traders on Both Sides

Context Plug-In and Application share a Plug-In 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 Definition and Framework 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 or via Product Trader mechanisms.*

Place traders for classes that are completely known within the 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 traders for classes that only the Plug-In can know, on the Plug-In side. Framework Application decides when these are used. Often the Plug-In trader is asked only for objects that the Plug-In has defined as available during Plug-In Registration.

Consequences

- ☺ Application and Plug-In can define a Plug-In 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 trader on Plug-In side is the `PlugIn` class itself, through which the application addresses all Plug-Ins uniformly. On application side, use factories instead of traders when the Plug-In creates the instances (domain objects).

Pattern 14: Plug-In Defines Subclasses

Context

Applications owns techniques, Plug-In knows the semantics.

Problem

How can the Plug-In deliver objects with semantic meaning, that have access to the technical infrastructure? Who creates them?

Forces

- Objects must know Plug-In semantics and use application techniques
- Only Plug-In knows the specific behaviour of the objects
- Application does not make use of specific behaviour

Solution

The Plug-In subclasses where the application defines superclasses, and expects the Plug-In to deliver instances that the application uses in a generic way. The application decides about the time of creation, and asks the Plug-In traders or factories for the instances.

Consequences

- ☺ Knowledge of techniques can be encapsulated in the superclass and kept within the application
- ☺ Plug-In subclasses contain the complete behavioural specifics
- ☺ Application can address all Plug-Ins alike
- ☹ Not all technical services are available to the objects

Implementation

Plug-In entry class

The Plug-In delivers during startup an instance of the abstract class (interface) `PlugIn`, that the application defines and addresses during the Plug-In lifecycle.

Use a Factory Method on Plug-In side in this case (similar to *Gamma+94*, but specific for activation technique of the Plug-In).

Plug-In private class

The Plug-In delivers various instances of defined superclasses, where the application's superclass encapsulates implementation issues and the Plug-In provides application know-how.

Use a Product Trader [*Bäumer+97*] on Plug-In side in this case.

Example

`GuiDialog`. The superclass from the application knows how a dialog is drawn, which basic widgets are available, and how the dialog accesses them. The subclass from the Plug-In knows where which widgets are drawn, and what effects their actions have.

Pattern 15: Plug-In Parameterizes Application Classes

Context

Applications owns techniques, Plug-In knows the semantics.

Problem

How can the Plug-In deliver objects with semantic meaning, that have access to the technical infrastructure? Who creates them?

Forces

- Objects must know Plug-In semantics and use application techniques
- Application knows the complete behaviour of the objects, i.e. the semantics are closed
- Plug-In defines the time of creation

Solution

The application defines constructors where all relevant domain knowledge can be passed as arguments. The Plug-In creates domain objects with construction arguments, and expects the application to perform common application domain and/or technical services with them.

Consequences

- ☺ Knowledge of techniques can be kept within the application domain class
- ☺ Creating specific instances is very convenient
- ☺ Factory mechanisms can be employed where necessary
- ☺ Higher amount of reuse, as behaviour is defined by application
- ☹ Subclassing is not supported, domain can not be re-opened

Implementation

Use a Factory on application side when the instance handling requires technical services that the Plug-In should not know about, like persistence.

Warning: Resist the temptation to broaden the 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 would 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 becomes very difficult.

If the Plug-In is expected to need specific object structures, but no specific behaviour, consider adding Composite [*Gamma+94*] classes to the application domain, that each Plug-In may parameterise according to its object structure.

Variant **Product Specification by Plug-In**

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

An `Alarm` object can be characterised completely by construction parameters like string resource ID and priority. All handling and logging mechanisms can be kept privately within the application.

Variant: `Rack` and `LoadList` for a chemical analyser need invariant physical properties of the particular analyser as construction parameters. These can be retrieved from the Plug-In properties.

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 [*Brown+99*], 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 own 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 which can live only in the context of their application.

Relation Framework-Providing 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 [*Johnson99*]. The Framework-Providing 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 several aspects. 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 Host Application. They are not freely pluggable, but limited to their domain and application. A key to Plug-Ins is the mutual contract that they fulfil, but that is controlled by their surrounding Host Application. While this approach is narrower than CBD, the range of applicability is broader. Plug-Ins even appear in embedded systems, while today’s Components are mostly limited to distributed enterprise models. Nevertheless, the cross section is visible, and when Component Based Development further succeeds, the mechanisms available for Plug-Ins will become more convenient.

Where do we go from here? - One future task is a pattern collection for Product Lines. Some preconditions are given here, but a Framework-Providing Application and a number of Plug-Ins do not make a product line. A successful product line implies reusable Framework-Providing 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 another result of our shared experience. Special thanks to John Vlissides for shepherding this paper before EuroPLoP 1999, to the workshop participants for their constructive criticism, and to Neil Harrison and Dirk Riehle for their guiding feedback on earlier versions of this paper.

References

Bäumer+97 Dirk Bäumer, Dirk Riehle: Product Trader. In: Pattern Language of Program Design, Volume 3, 1997

Brown+99 Kyle Brown, Philip Eskelin, Nat Pryce: Component Design Patterns. Work under construction at http://c2.com/wiki?ComponentDesignPatterns

Gamma+94 Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison-Wesley 1994

Johnson99 Ralph Johnson's framework home page, http://st-www.cs.uiuc.edu/users/johnson/frameworks.html

Köthe98 Ullrich Köthe: Design Patterns for Independent Building Blocks. In: Proceedings of the Third European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1998)

Marquardt98 Klaus Marquardt: Patterns for Software Packaging, Installation, and Activation. In: Proceedings of the Third European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1998)

Noble+98 James Noble, Charles Weir: Proceedings of the Memory Preservation Society. In: Proceedings of the Third European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1998)

OCF98 OpenCard Framework. General Information Web Document. OpenCard Consortium and IBM, 1998 (retrieve via http://www.opencard.org/)

Pree97 Wolfgang Pree: Komponentenbasierte Softwareentwicklung mit Frameworks. dpunkt, 1997

Roberts+97 Don Roberts, Ralph Johnson: Patterns for Evolving Frameworks. In: Pattern Language of Program Design, Volume 3, 1997

Sommerlad99 Peter Sommerlad: Configurability. In: Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1999)

Szyperski98 Clemens Szyperski: Component Software: Beyond Object-Oriented Programming, Addison-Wesley 1998

Szyperski99 Clemens Szyperski: Components vs. Objects vs. Component Objects. In: Proceedings of OOP 1999

Völter99 Markus Völter: Pluggable Component. In: Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1999)