Klaus Marquardt
Dräger Medizintechnik GmbH
D-23542 Lübeck, Germany

Phone +49 (0) 451 882 - 3314
Telefax +49 (0) 451 882 - 2727
EMail klaus.marquardt@draeger.com

Physical Patterns

Klaus Marquardt

Abstract

Patterns have emerged as an efficient way to communicate solution proposals. Most software patterns up to now focus on logical design issues. This article presents physical design patterns, that partly employ existing logical patterns.

The presented patterns deal with limited physical resources, and framework development. "Advent", combined with the logical "Light'n'Heavy", facilitates online installation of software components. "Plug-In" describes how large logical components are physically integrated. "Suppengemüse" describes components satisfying a variety of technical interfaces at the same time.

About Physical Patterns

- What are physical patterns?

  Physical patterns are patterns that go beyond the scope of logical design issues, and focus on physical design instead. Small physical patterns are often called "idioms" and are closely related to a specific language. The patterns presented here go beyond the scope of a particular language, and even beyond the scope of classes and their collaborations. They require existing design patterns as a prerequisite, and combine these to facilitate physical aspects of architectural range.

- Why are they called “physical” patterns?

  Because I could not think of a better name yet.

  Although these patterns have a logical context that they do not really leave, they combine logical aspects in order to facilitate medium and large scale physical design issues. Physical patterns result in files, dynamic link libraries, and released components consisting of a number of files, similar to Design Patterns resulting in classes and collaborations.

- Do they depend on the development or runtime environment?

  Some of them do, as described in the patterns themselves. Issues of file separation and process context are an essential part of them.

Known Uses

The author has seen these patterns used in proprietary systems at his work. These systems were applications and frameworks in chemical and medical environments, covering aspects of both technical and business information systems.

Acknowledgements

I would like to thank my colleagues at sd&m and Dräger, among them Gerhard Koller, Thorsten Petersen, and Lars Pickert, for their fruitful discussions, open minds and new ideas, and for making the ideas become implementation. Special thanks to Jens Coldewey for reminding me that there are things to learn, even outside of the current project, and to Neil Harrison whose questions and suggestions turned the original ideas into pattern descriptions.

References

GB Grady Booch: Object Solutions, Addison-Wesley 1996

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

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

RM Robert Martin: Designing Object-Oriented C++ Applications Using the Booch Method, Prentice-Hall 1995

Light’n’Heavy

Possible other name: Laurel and Hardy

Context

In a legacy system, the established design causes performance problems. An often used class is designed very huge, with a large but consistent set of responsibilities, some of them requiring a lot of member variables. Some of these member variables do not vary so much; many instances share identical contents of them, although the contents is not trivial and constant for all instances. You can define groups of instances (populations) with that common part, and the great majority of your objects belongs to one of these groups.

Problem

Due to changes in requirements or environment, your system performance became unacceptable. How do you make changes to the original design to improve performance?

Forces

The system shall be fast, so design must contribute.

You want to keep your domain objects definitions independent on performance issues and tuning options. You try to avoid that implementation changes migrate through large parts of your system.

You may already have external users of your object, and cannot afford to make significant changes to the interface, and none to the contract.

Solution

Place the unavoidable costs of construction and destruction at a point in time where they hurt least.

Separate the original class into an expensive, quasi static part (Heavy), and a less expensive one (Light), while ensuring that the light part is much more frequently used. This works best in a situation where you have a large part common among populations (groups) of objects, and a few specific members (like a finite state machine) for each single instance. Light and Heavy stand in a N:1 relation.

Identify candidates to place in the Heavy class by asking: Which are the most expensive members in construction? Go for the ideal case, trying to create everything just once: Can they become static? (In the given context, it is not possible to really use static members; members are shared quasi statically among larger populations of instances, but not among them all.) If not - do they share the same contents for a large number of instances? Is it likely that further instances also share this contents? Those can be put in the Heavy class.

After having identified member variables for Heavy, try to find the functions: Which member functions do refer to these variables only?

Clients to the original class now become clients to the Light part. Light holds a reference to its Heavy partner, and delegates all requests related to non-individual members to it. The Heavy partner is a fixed part during the whole lifetime of the Light instance. Before creating a Light object, its corresponding Heavy class instance must have been created.

Example

Consider a building observation system that monitors for violations of access rights, temperature limits, and doors left open. All violations are announced in a central office both visible (with a string depending on the language selection) and audible, depending on their priority setting.

We identify three different users of the violation: a creator, an observer, and a configurator. Thus, we decide to model a common base class for all violations: Violation. An ad hoc class design could look like this (note that there may be many more configurable items in the final model):

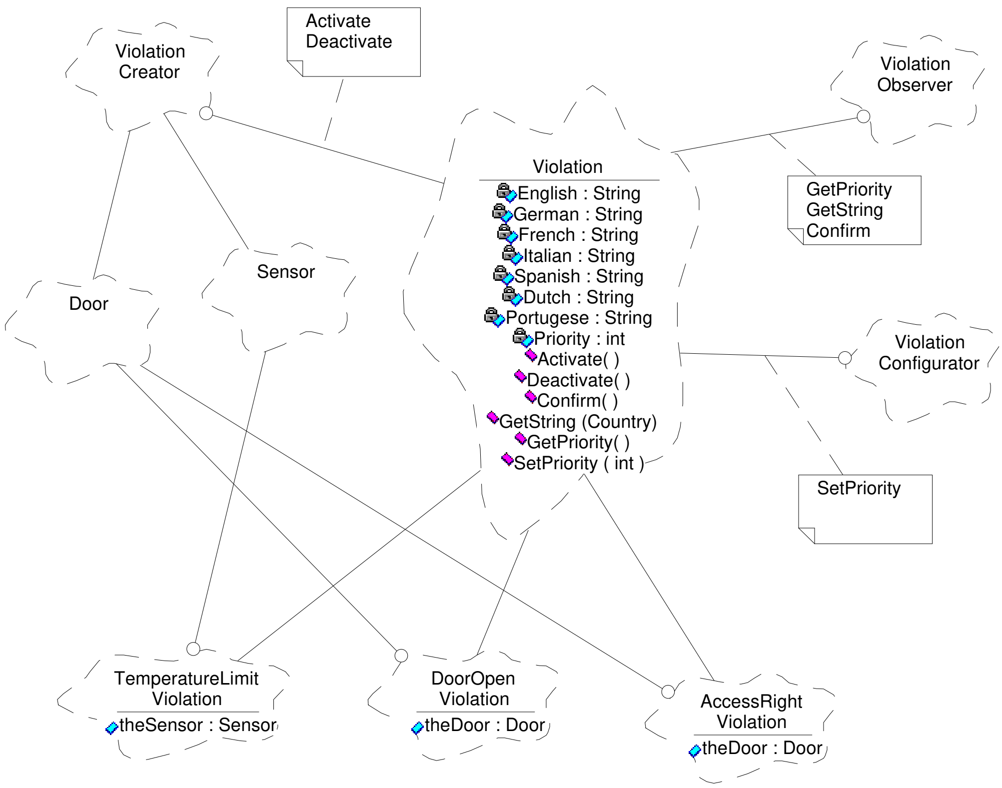

Figure 1: Ad hoc design for the building observation system

Each violation goes through the same life cycle:

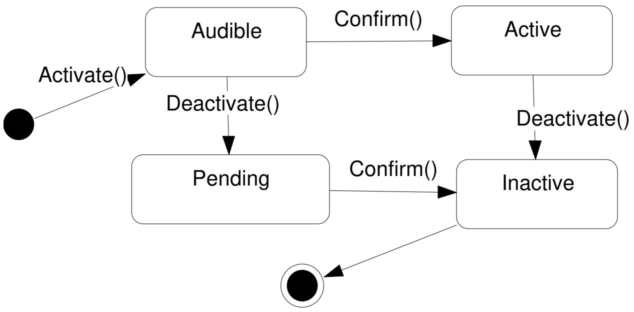

Figure 2: Violation finite state machine

In second release, it becomes required to archive all violation occurrences, together with information about their occurrence time and the confirmation time. We get more users of Violation: the archive, the archive cron, and the archive observer.

Now a design flaw becomes obvious that we already made in the initial design: class Violation became too large. If each instance is to be archived, the strings, priority (and all the other settings) are stored multiple times. But we are just interested in storing the time of activation and confirmation, and the specifics (which sensor, which door). Thus, we split Violation into a Light and a Heavy part. The Light part that maintains the life cycle and keeps the client interface stable, by becoming a Proxy to the Heavy part. The Heavy part keeps all settings, and is stored only once per violation class.

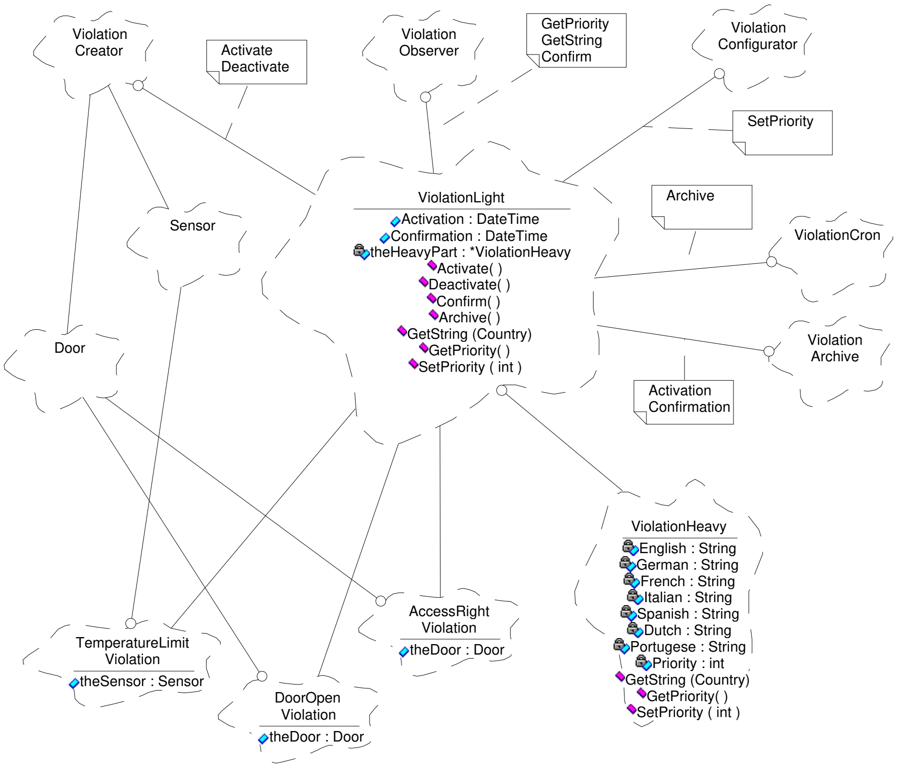

Figure 3: Design for the building observation system with archive

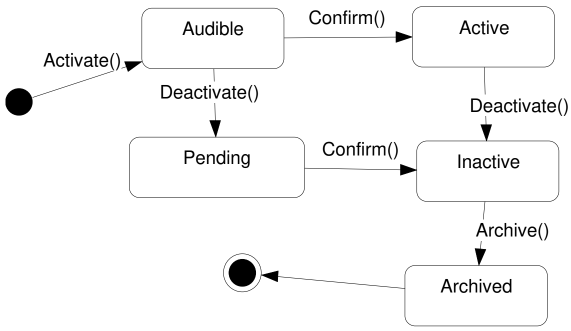

Figure 4: Violation finite state machine with archiving

Consequences

You managed to implement a major design change with only minimal[1] or no changes to the external interface. Except for additional private members and the need of changing the application initialisation, the interface and outside responsibilities of the light class is identical to the original class. Existing users are likely to need recompilation though (depending on you runtime environment).

Your design change gives you more options to tune your systems performance. Splitting costly classes into a Light'n'Heavy couple makes frequent creation and destruction, and storage of similar objects less costly.

Application initialisation takes more time: You need to create the Heavy object, before creating the first Light object referencing it. Depending on your context, you may chose lazy instantiation or instantiation of all Heavy objects during startup. Lazy instantiation distributes the load among a larger time frame, while startup instantiation may take some time, but allows you to access the Heavy part directly if needed (see Implementation).

The domain objects definitions do not reflect your internal design for performance, except for an additional private member variable. In case you access the domain objects through factory mechanisms already, the change is invisible.

More classes are to be developed, tested, and maintained.

Implementation

For the implementation, it is essential that the Heavy instance is already available when the Light is created. You could think about searching for an appropriate, previously created Heavy while constructing a Light, but that would not solve your performance problems - you would notice them even harder during creation. You need to do a thorough domain analysis, identifying all the Heavy objects that your application needs. These Heavy objects are similar to Singletons (GOF, 127), but you have a defined number of instances with different

[1] You may need to insert a private member variable. Depending on the development and runtime environment, and the existing design context, your clients might have to recompile.

parameters.[2] You can create them either at application startup time, or "lazy" on demand when the first Light instance referring to them is created. The creation strategy decision depends on where you can best accept the (physical) costs (see pattern Advent below).

Sometimes your domain analysis can not be completed, or might even prove that you cannot identify all Heavy instances in advance. (This could be the case when they are created by a Plug-In, see below.) Then you need to create the remaining few during Light creation. For creation performance, be prepared for some redundancy here due to Heavy duplicates, and consider a way to minimise the Heavy duplicates.

Variation

If you have additional forces into maintaining stable interfaces to existing clients, but offering more services to new ones, you could consider leaving a narrow legacy system viewpoint: Occasionally you might be able to fulfil the additional requirements, if you expose both the Light and the Heavy part to clients.

Changes to the external interface are larger then, but existing clients (accessing Light only, that still maintains the whole original interface) would still not have to change. Future clients could make use of the extended lifetime of the Heavy part. But note that future developers will always feel insecure about which class to use, as the interfaces are partly identical and class responsibilities are not clearly visible. In the above example, you would gain the ability to change Violation settings even before an actual Violation occurs.

Related Patterns

Flyweight: How is Light'n'Heavy different from Flyweight (*GOF*, 195)? Compare Light to the Flyweight client, and Heavy to the Flyweight. The same idea applies for their intrinsic and extrinsic state. Differences are in intent (Heavy is really a large class), that the Heavy needs no subclassing or factoring, and that the client to this pattern is a client to Light, not to Heavy. Flyweight and its clients do stand in a N:M relation, not in 1:N as Heavy and Light do. With the implementation alternative, the difference between both patterns shrinks.

Advent: The Registration phase of Advent can be tuned in size compared to Activation. The more Heavy instances you have, and the earlier you can choose the time of their creation, the higher is you potential for shifting costs into Registration (or at least into Loading). For Advent, it can help to use the variant of exposing also the Heavy part.

Plug-In: When the Light'n'Heavy pairs are hidden as part of a (legacy) framework, maybe even through factory mechanisms, the Plug-In clients are least affected by this design change. When the Light'n'Heavy pair is part of the Plug-In, you are no longer in a legacy situation; but the design change has only local effect now, and can not increase performance of other framework clients.

---

[2] Which appears to be a common extension to the Singleton pattern. *GOF* mentions subclassing as a solution, but you can also parameterise the Singleton, similar to what you can to with printers: One printer is always the active one at a time (classical Singleton); in special applications, you could use many of them - but the one in the north-west corner of the 4th floor exists only once (parameterised Singleton).

Advent

Possible other names: Four Phases; Clover; Stairway (to heaven)

Context

A System that is not closed but extensible by complex attached physical components, represented through appropriate software components (drivers). When these components are needed, they are often needed immediately. Which components are to be activated when cannot be foreseen, but installation is done in a previous, separated step. Some components are mutually exclusive. System resources are limited, and the systems load may be high. Not all services of the system may be available at a given time. System availability must be 24h/d and 365d/a, except by explicit interrupts initiated by the user.

Problem

Starting up an attachable component takes significant time. You need to be prepared for this, because the system users demand high responsiveness. In a security system for example, you would not want any switching to cause a break in observation, even if it were only for a few seconds.[3]

Forces

Users demand high responsiveness.

You cannot just activate in advance everything that might come.

Actions caused by attaching or removing a physical component must not disrupt the system.

You cannot do too much at the same time in the background.

You must be able to prepare for new components at any time.

You would like to use standard tools for your software installation wherever possible.

Solution

Separate the following four phases of installation and activation that precede actual execution:

1. File Copy

   The files of which the attachable component consists, must be copied to the system.

[3] But without predictable observation leaks, we would have missed some fascinating movies…

2. Registration

   The organisation objects of the component are known in advance, and can be registered (e.g., stored to database) long before the component really becomes active.

3. Loading

   Loading and executing code need not be done at the same time. You could improve responsiveness of the system, if the necessary code would already be loaded into memory before the execution is really demanded. You can also create instances of Singletons here, and retrieve your working environment.

4. Activation

   This is the real thing. Do this when the user finally demands the component working.

In most environments, the phases File Copy and Registration, as well as Loading and Activation, are typically done together, but there's no real need for this. You are able to shift costs to more convenient times when you go through these phases one after another.

Separating File Copy from Registration allows you to prepare system extensions while the system is running, and service personnel may leave the right time for Registration to the customer. Separating Loading from Activation allows you to preload drivers for anticipated components during non-critical times, and minimise visible response times, without causing too high costs when other components are to become active.

Example

Take another look at our example. We are extending our system to develop a building observation central, maintaining security services for a large number of buildings. Some of them are always centrally observed, some during known times of the day; others on urgent requests. All buildings have different violations to report; unlike others, a museum might want to observe humidity and light. We need to be able to connect to these buildings (add their drivers to the system) while the system is active and under high load.

Consider we have found a way to pack all Violations that a particular building adds to the system, in some kind of Plug-In (see next pattern). We would demand from a new customer that he delivers all needed files before his building can become observed (in real life, we would support him to create the files). These files must be available at our central server (Phase 1: File Copy). Due to the split into Light and Heavy, installation is not finished by copying files. We can not use any Violation before its Heavy part is available, but we can make Heavy available before we use the Violation, for example by writing it into the database (Phase 2: Registration). Depending on the workflow of the central observation (switching observations, or parts of them, depending on time or certain events), it might be helpful to have larger parts of the code preloaded before it becomes active, or simply separate these steps to avoid peaks loads (Phase 3: Loading). Finally, as soon as the building must be observable at the central station, its driver is activated (Phase 4: Activation).

Consequences

Tuneable system responsiveness without logical design compromises.

Note that this one is not foolproof! If the proper steps are not taken beforehand (for example no preload took place before activation), the user will still experience visible delays. When you preload the wrong components, performance will degrade. Worse, when the Registration phase did not take place, the user might experience that the expected functionality is not available at all.

Increased development effort. All extensions need an individual check for shifting load between the phases, and need to support them as separated. You need specific knowledge about your system to fine tune your load profile, and serious consideration about what you allow your end user to change.

Phases File Copy and Loading, which must be executable in the background, are separated to allow for a decreased system load during initialisation. File Copy can be done any time, Registration can be shifted to the most convenient situation. Separating these phases can also enable you to prepare for the addition even when the system services needed for complete Registration are not available.

Phases Registration and Activation require administrative or user interaction in most systems. They represent the visible "installation", and visible “starting”. Depending on the forces in your system, a visible delay may be acceptable here.

If phase Activation is initiated by user interaction, it may be allowed to cause visible delays. If activation has to be automatically, putting maximum costs into Registration could help you decreasing Activation costs or costs during execution (construction of persistent objects, “proactive instantiation” instead of early or lazy instantiation)

Your (physical) solution is bound to your environment because you need to employ a lot of specifics like DLLs, databases, etc.

You cannot use standard utilities as a complete solution for your software installation: these tools can in principle not offer you a way to initiate the Registration, or separate Loading from Activation.

Implementation

Which phase should carry which physical execution costs, depends on your load profile. So a first step to implement Advent is to analyse your system load. Try especially to analyse workflow dependencies in the system load, and whether you can define non-critical low load times. Into these time windows, you can put the Registration and Loading phases, and make them as expensive as you can to take load from Activation and actual execution.

For designing the load costs, it helps to think of the execution of specific functions as a fifth phase (according to the pattern name, this would be Christmas). How can you take costs from one phase, and place them into another?

These decisions increase the costs in File Copy:

- Using many, and large files.

  To use information in files instead of increasing the (sometimes unspoken) context of contract between different application parts, is an important decision in itself (see also pattern Suppengemüse below). Splitting your needs into multiple files allows for sophisticated partial Loading and Activation, decreasing the costs there.

These decisions increase the costs in Registration:

- Populate the systems database with the objects you would need to make persistent during execution, as far as you know them now.

  For objects linked to others that originate from other software components, this is a clear advantage.

- Use Light'n'Heavy.

  You can build all your Heavy instances in advance, and store them into a (object-oriented) database ("proactive instantiation"). In later phases, you would have to retrieve them - the cost effectiveness here depends heavily on your database and caching mechanisms.

  This works best if you have exposed the Heavy part to clients (see variant of Light'n'Heavy). If you have not, you are forced to find a work-around (like creating an additional registration application for the legacy client, that issues all Light instances of that client once during registration so that you can create the Heavy parts yourself. This application becomes an additional part of the Suppengemüse delivery).

- Make parts of your application configurable, and create the configuration information during Registration.

  Especially if flat file access is not feasible, you can populate your configuration database during Registration.

  As a side effect you can possibly access and influence the configuration even while the corresponding code is not executed. This may be highly desired while testing or debugging, and highly undesired with software embedded in a product.

These decisions increase the costs in Loading:

- Load all files in memory.

You take costs from Activation and execution here. The Loading costs may be distributed over time when you can load the component part by part.

For the following activities, you can choose whether to take them into Loading, Activation, or execute them later:

- Use Light'n'Heavy. Create your Singletons statically.

  You can create all your Heavy instances and Singletons in advance, before they are used ("static instantiation"). Alternatively, you could created them when first needed ("lazy instantiation").

- Allocate memory.

Your choice depends on your system, on the results of your workflow analysis, and on further forces. In some real time environments for example, you might not want to allocate memory in the mid of execution, to avoid failure in critical situations. Every activity that you manage to place before execution, takes processing load from there, and makes your load profile more predictable.

Related Patterns

Light'n'Heavy: Link this with the Light'n'Heavy pattern, which helps you to put costs into phase Registration (which would be the cheapest phase if you do not employ this pattern).

Plug-In: Advent is especially useful for dynamically loading Plug-Ins into an already running system.

Suppengemüse: Use this pattern to define the number and size of files which must be copied during File Copy phase.

Plug-In

Other possible names: (none)[4]

Context

You are developing a framework which must be able to dynamically use specific implementations (components), that are employing the framework and adding semantic knowledge to it. You cannot know about the component except for its interface (the component may be installed separately), but you must load it at runtime. All extensions are determined during system runtime.

You are using a computer language with static linkage, and an operating system with some dynamic linkage facilities (DLL). You do not want to expose process context to the framework clients, so clients can only contribute logical components. Due to performance restrictions, the process context (number and priority of processes or threads) may be a tuning option, that you cannot give away to a Plug-In developer building an application.

Problem

You need a way to make use of the extension, and allow the extension to make use of the framework, without a mutual logical and physical dependency.

Forces

The framework must not depend upon an extension component, only on its interface (that the framework defines itself).

A component must not depend upon an implementation solution within the framework, or upon process context.

Solution

Design your framework consisting of two parts: one that makes use of (calls) the functional extension offered by a Plug-In, and one that the Plug-In may call. Publish the interface you expect the Plug-In to fulfil, and the interface you offer to it. Design your specific implementation as a Plug-In and place it in a DLL.

On a very global level, Framework and Plug-In look like this:

[4] The author has seen names used like Driver, Extension, and Socket. All these names do not work, and were a cause of misunderstandings among engineers of different cultural backgrounds.

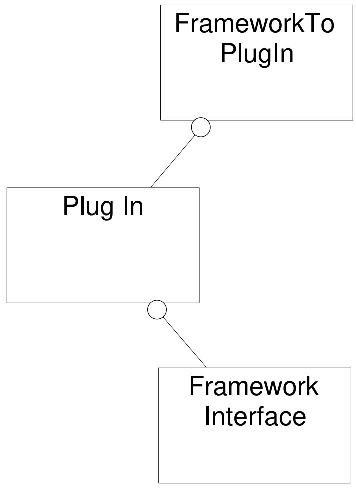

Figure 5: Principal physical components of Framework and Plug-In.

Take a look at the logical entities that are within the physical components:

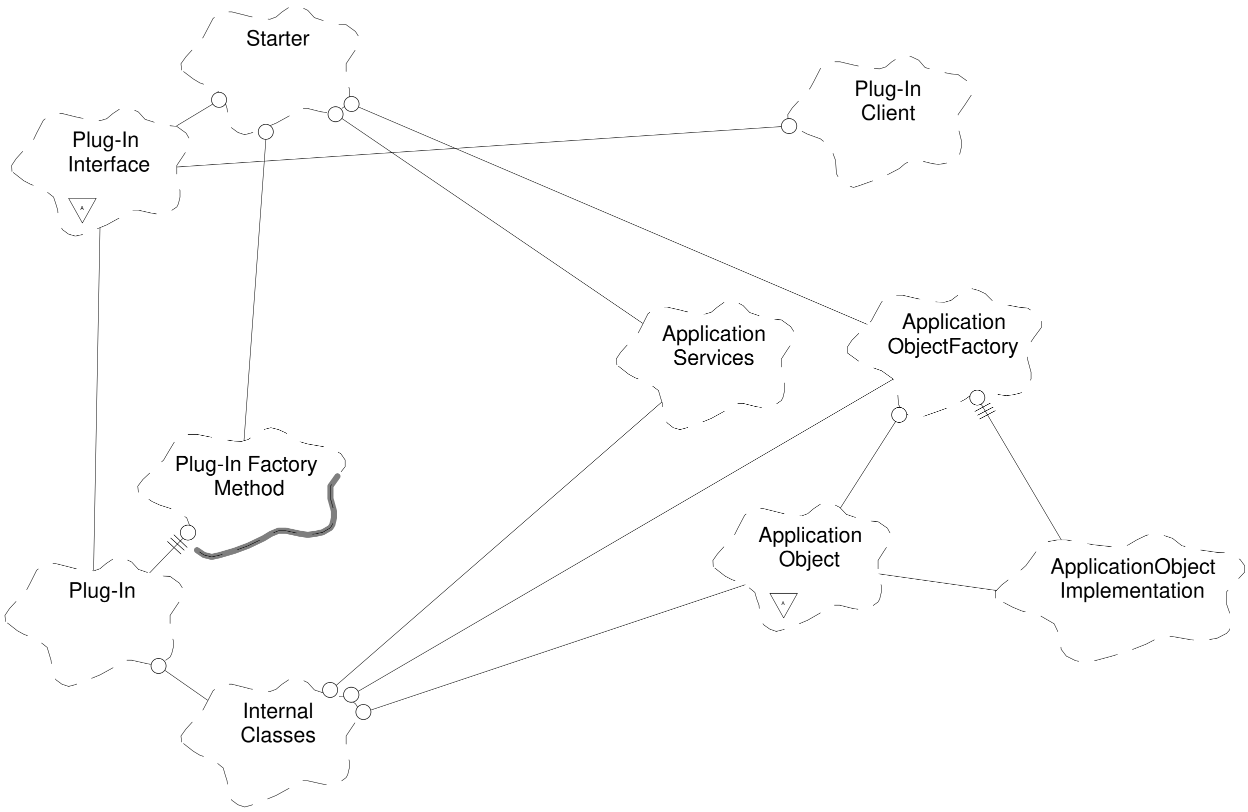

Figure 6: Plug-In and the contained logical components

The FrameworkInterface component provides general services for the application, and factories for classes whose implementation must be hidden from the Plug-In.

The Plug-In DLL provides a Factory Method (similar to the *GOF* Factory Method, but implemented as a free static function instead of a class member) that returns an instance of the expected interface class. (Plug-In and Plug-In Interface become parts of a Bridge pattern here.) The extension class completely encapsulates what is inside the Plug-In component.

The FrameworkToPlugIn models the whole extension as one class, and publishes the abstract interface class as well as the factory method it expects. It provides a process context for execution, invokes (or connects to) the FrameworkInterface component, loads the appropriate Plug-In DLL and calls the Factory Method to create an interface instance. This instance is provided with references to the FrameworkInterface manager classes.

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

You need to make decisions what the framework shall offer, and what the Plug-In has to provide and implement itself. Putting more services and application domain objects into the framework allows for a larger amount of reuse, but narrows the possibilities for extensions and may limit the reusability. Make an extensive domain analysis, to ensure you understand your domain well enough to make tough decisions! If you fail here, you will either have large development effort with the Plug-In's, or you might not be able to fulfil expectations of future Plug-In's.[5]

Example

Our building observation system must be able to connect to a variety of different buildings, equipped with sensors and systems from different vendors. We can remotely connect to them, given that the protocol and the internals are known.

All these vendor specific system have things in common - this is why we have our market at all. For example, all do activate violations. And there is only a limited number of violation kinds - they may be based on a mere sensor signal (door open), on a sensor limit violation (temperature high); all this combined with conditions like a timer and a persons ID. (Allow yourself some time to analyse the violation kinds completely - for this situation it will become essential to get it closed!)

The violation kinds, and the violation related services like announcing and archiving, are placed in the FrameworkInterface component. The knowledge about the vendors protocol, and how to translate it into violation objects, is placed into the Plug-In. The definition of expectations towards the Plug-In, including technical issues like connecting and disconnecting gracefully, are placed in the FrameworkToPlugIn component.

The design in Figure 7 shows parts of the framework definition and implementation. The framework implementation can make use of the Plug-In: Note the error handling facilities as common services, that are at the same time clients to the Plug-In, addressing it via its abstract definition.

[5] Do not take this too serious, do not be paralysed. Whatever you do might turn out to be wrong in the long term. Just make sure that you know what you are doing. (see *GB*, Rule R79)

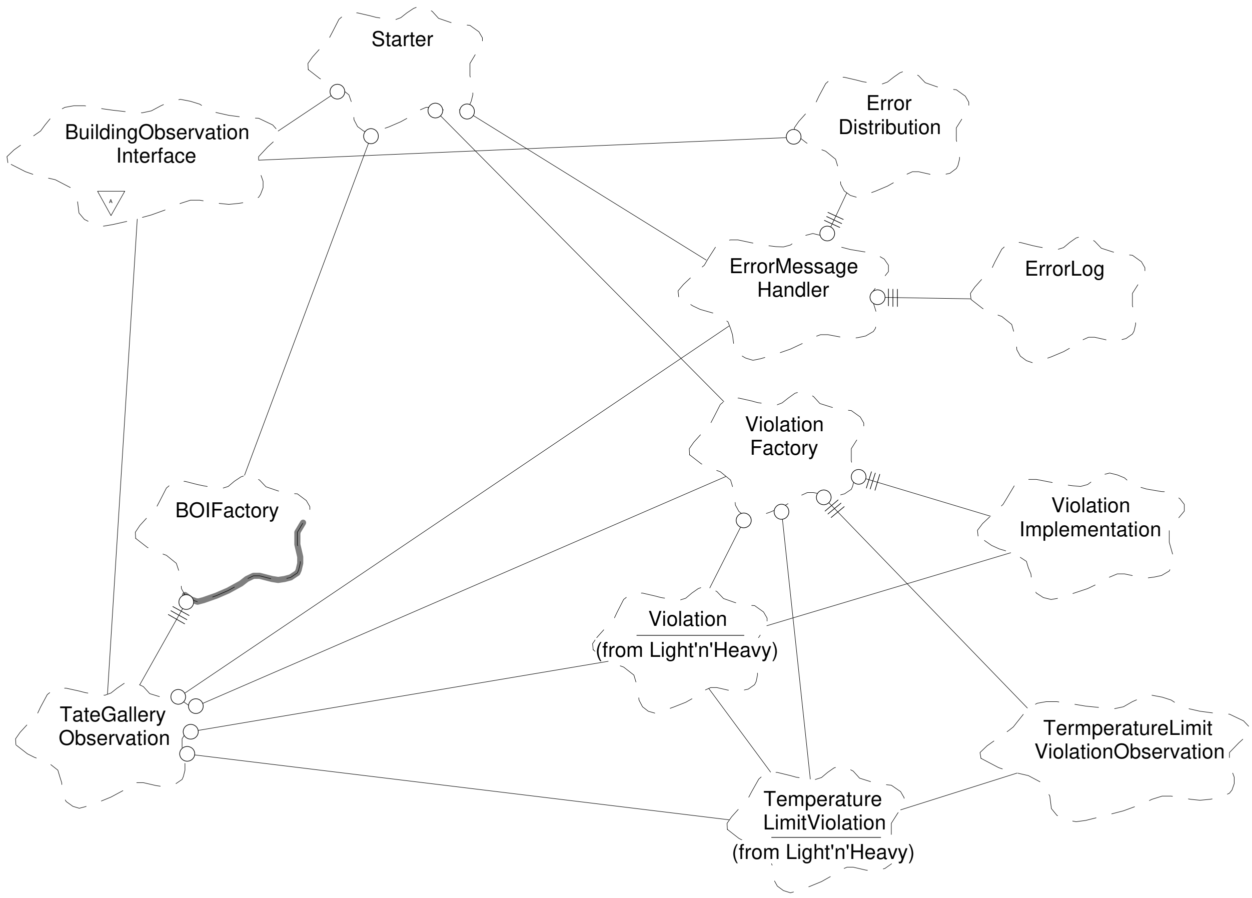

Figure 7: Framework and Plug-In components for the building observation system (simplified)

Consequences

Framework and Plug-In can use each other without knowing each other's internals.

A bunch of classes must be developed just for dependency management.

Having factories in the Plug-In enables the framework to use an implementation unknown during compile and link time. It can be loaded an unloaded during run time. But the factory prevents using framework services that can only be reached via Mix-In and multiple inheritance - persistence in OO databases is a common example.

Having factories in the FrameworkInterface enables the framework to hide its implementation to clients (Plug-Ins). The classes coming from the factories cannot be extended (via subclassing) by the Plug-In, because this would require exposing the implementation, or large parts of it. Thus, the application domain defined in the framework must be designed carefully to remain stable and sufficient for a large number of Plug-Ins. Redefinition of the domain for a "special" Plug-In would affect all other Plug-Ins as well.

If you fail to identify all subclasses you want to offer via a factory on framework side, you will not be able to offer them to clients later without changing your external interfaces. Where you are insecure about the quality of your domain analysis, you should consider removing the factories and exposing your technical details.

Implementation

Take a very careful look at the dependencies between framework and Plug-In, to avoid all mutual dependencies. (*RM* and *JL* give a good overview on dependencies, and a great deal of advice how to manage them.) Figure 8 shows appropriate fine grained class categories, that depict those parts of the framework that all other parts, and the Plug-In depend on: the Plug-In Definition, and the FrameworkInterface. It also contains the client relation from FrameworkImplementation to Plug-In Definition, that the example added to the solution.

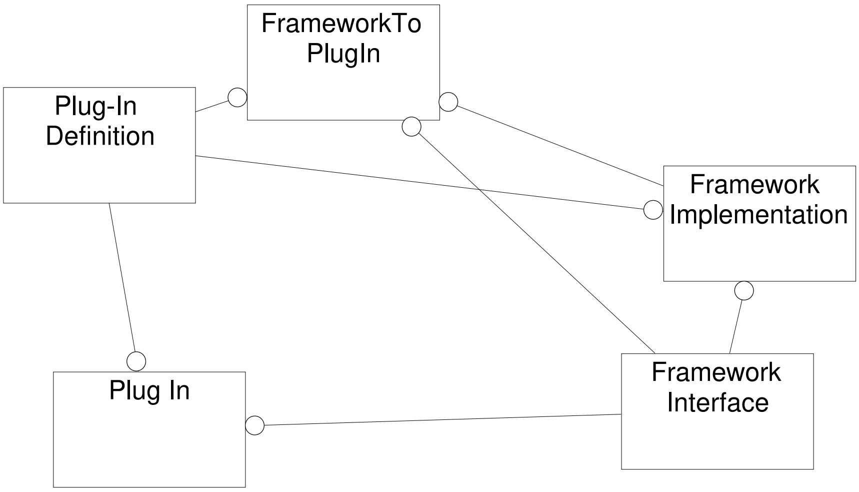

Figure 8: Dependencies of Framework and Plug-In components

The DLL may have arbitrary other exported functions, but they are not referenced by the framework. This leaves a good deal of Plug-In implementation up to effort and personal taste.

The framework may use Proxies (*GOF*, 207) for manager objects or application object implementations that are not running in the same process context as the Plug-In. Often the abstract Plug-In class is an Observer (*GOF*, 293) to commands and application objects. Other useful patterns for communication from the Plug-In to the framework are Command (*GOF*, 233) and Strategy (*GOF*, 315).

Do I always have to use the Bridge pattern, when I want to hide implementation and break dependency cycles? Use Bridges for the Plug-In and the application object abstractions. Use compiler firewalls for the Factory Method, and for the framework manager classes when a Bridge would be overdone. Techniques how to use language specific constructs as firewalls are described in *JL* (for C++).

As a Plug-In designer, realise that a Plug-In implementation can contain much more than a single class, so you can engineer your Plug-In also under consideration of design by contract, with reuse in mind, etc. As a framework designer, be aware not to load too many responsibilities into a single Plug-In. Suppengemüse (below) can help you to divide, and allow your clients to clearly overlook the parts they have to build.

Related Patterns

Advent: Plug-In is the software component that may be preloaded, and becomes activated.

Light’n’Heavy: If Light’n’Heavy is used by the framework to separate the Registration phase of Advent, Plug-In has to retrieve registered objects instead of creating them. This can be hidden, but requires an access layer on its own. Plug-Ins may also use Light’n’Heavy, this can help locally to gain performance.

Suppengemüse: Typically, a Plug-In is not the only thing that you need to make an attachable component work. You may need several of them, and you may need additional files for specific services (e.g., localised texts).

Suppengemüse[6] (Vegetable Soup)

Possible other names: Bouquet, Medley, Besteck[7]

Context

You are developing a framework that is prepared for complex extensions. Lifetime of the product or product family in the market is very important, and so becomes your framework architecture.

Problem

Lifetime depends on the flexibility that future framework users experience. This flexibility may include specialised application object behaviour, human interface, localisation, and communication. These parts cannot be covered completely by a Plug-In, or a protocol alone (at least not by a single Plug-In class).

Forces

Rely on existing standards for technical interfaces. You do not want to invent them all on your own.

As technology evolves, the framework will be used in ways that you do not imagine today.

Your system is safety or security relevant, and needs certification that is not broken by future system components.

Your system, and in particular its interfaces, needs to be stable to lengthen its lifetime.

Marketing demands quick reaction times, and short development cycles.

Solution

Define a number of technical interfaces to be accessible by the future framework user, and functionality your framework expects from a client. Each technical interface and functionality may correspond to a separate file that the clients deliver as part of their "Suppengemüse" package.

How can I identify the files I require from a client? Start by identifying the expected functions first, and then the technical interfaces. How do I find out which interfaces to publish, which standards I can use, whether they are sufficient and appropriate? A great deal of what to

[6] The German "Suppengemüse" denotes the vegetables that are bought as a basis for a soup. It is a common term. At the greengrocer, you always get the same vegetables (celery, carrots, leek), always the same amount of them, and they are always bundled together.
[7] The German "Besteck" denotes a set of tools used together: knife, fork, and spoon.

publish depends on the clients you expect. Are they developed within your team, or within your company? Do you subcontract parts of your development, or do you even publish all your interfaces and let other vendors build Plug-Ins to interface to your framework? The more control you have, the broader your interface can and should be.

Expected functions are often delivered by a Plug-In. This Plug-In DLL makes up your client's first file. Can you get along with one Plug-In for the client? You could have multiple of them:

- for communication with an external device; this is most likely covered by a single DLL;
- for providing parts of your user interface, like windows; each client specific window could make up a DLL on its own, or you combine all of them into a single DLL;
- for adding semantical knowledge to the system, for example algorithms that process the domain application data in a client specific way; these algorithms could each make up a DLL on their own, or you combine them into a single DLL.

Now you have identified files you would like a client to deliver according to the functionality you expect him to provide. The Plug-Ins all connect to a specific FrameworkInterface, that provides domain objects, parts (or all) of your graphical class library, and a potential large number of other services.

Some of the described Plug-Ins will tend to look like swiss army knifes. Often the FrameworkInterface is hard to understand because you spend effort in hiding implementation details, and preventing misuse. You need to avoid an uncontrollable interface by allowing framework users to access the framework internals (operating system, database), but you should not restrict framework users beyond necessity. There is a way out.

First, ask yourself: Are there foundations that the framework is built upon, that you can seriously consider stable in future versions? Class libraries are a perfect part to check here.

Second, take a look at the technical interfaces. Start asking: Are there requirements for nice little things, that are loosely coupled to the application domain? Especially the user interface domain can be full of them: you may have to show icons, animation movies, and playback sounds. Try to deal with them by using standard interfaces. For this you need both, an analysis of your application and its domain, and a knowledge of available standards and their expected stability.

Quite a lot of standards are available for user interface topics. Without exposing your class libraries to clients you cannot control, you can ask them to provide a file for an icon, a TIFF image, a MPEG movie, a WAV sound, and a number of localised string data in resource files. Try to put as many of the required features into these standard file formats. Each of them is small enough to be understandable, and each client's file is small enough to be implementable. This lightens your Plug-In Interface and helps for concurrent development of the client's delivery.

Example

When buildings are observed at the central office, for quick identification of a violation identification it is helpful when the acoustical violation announcement allows for immediate recognition of the affected building. Thus, you may want to add a specific sound as a part of the driver software for each building. It is most convenient to use the defined .wav file format here, and let the software driver consist of one additional file.

Consequences

You broaden the extensions interface, but split it up into manageable parts, i.e. standardised technical interfaces. Using standards here increases your framework interface stability.

You can enhance the number of interfaces that you offer to future clients, allowing for new ways of using your framework. Existing clients might be provided with a default in those interfaces that they do not cover.

Parts of the technical interfaces do no longer lie in your hand. You have to document or even implement restrictions if you anticipate changes in the interfaces, or if the user must not access certain parts of the interfaces.

Though each single file of the clients delivery gets as simple as possible, such a framework client software becomes a complex deliverable, consisting of multiple or even dozens of files.

Implementation

Sometimes your domain is well defined, and has well thought analysis public available. This common knowledge might have resulted in object models, or in application level protocol definitions.[8] You can combine using these standards together with the technical interfaces: Clients can offer more stability and user visibility at the same time, if they do not rely on one of these standards groups alone.

Take care to version and release all the files a framework client delivers together as a whole. This goes even beyond scope of development.

And, try to find a name for that delivery. Suppengemüse might not be appropriate after all.

Related Patterns

Plug-In: The Plug-In DLL(s) is/are usually the major part of the software Suppengemüse delivered for an attachable physical or logical component.

Advent: Separating the Registration phase often requires an additional registration application. This is delivered as one of the files within the Suppengemüse.

---

[8] Examples of standardised application level protocols are ASTM or HL7.