Architectural Development Pattern

Jorge Luis Ortega-Arjona and Graham Roberts
Department of Computer Science
University College London
Gower Street,
London WC1E 6BT, U.K.
{J.Ortega-Arjona, G.Roberts}@cs.ucl.ac.uk

Abstract

The Architectural Development Pattern allows one to consider a software program or system as a whole, to be designed and composed by different kinds of elements, which make it capable to be safely changed, adapted, and modified during its lifetime.

1. Introduction

The Architectural Development pattern, based on the concept of the Layers of Change in Software Architecture [OR99], can be used as an approach to the study, design and construction of maintainable software systems in general. It provides a base to understand how software systems change or evolve, by expressing and organising it as different subsystems as “layers of change”. These layers are defined according with the specific order in which generally they are designed and implemented. They also reflect the different rates of change that parts of a software program experience during its lifetime.

Patterns were originally developed for building architecture [APL77, TWoB79], and adopted later for software development [POSA96, PLoP94, Gab96, GoF94, PLoP95]. Their use is spreading in the software community to the benefit of those undertaking design and implementation tasks. However, other design practices can also be useful for software development. This can be probably the case with the layers of change. In fact, the layers of change, in the context explained by the architectural-based development, can be analysed as a pattern, or perhaps a meta-pattern, for the use of software patterns when constructing any complex software system.

Trying to apply this concept to software architecture though an analogy, it is necessary to make some basic assumptions, like what would be a “building” in software terms, who are its occupants or inhabitants, and how they are able to change and modify their “software buildings”. Richard Gabriel proposes an interesting analogy when introducing the concept of habitability in software:

> *“Habitability is the characteristic of source code that enables programmers, coders, bug fixers, and people coming to the code later in its life to understand its construction and intentions, and to change it comfortably and confidently”* [Gab96]

Originally, the word habitability is related to issues about buildings and the quality of living in them. Nevertheless, Gabriel applies this word to explain an important characteristic of software, which enables programmers and developers to “live” comfortably in and repair or modify code and design. Gabriel applies the concept of habitability as an analogy between software and building, comparing a program to a New England farmhouse, which slowly grows and is modified according to the needs and desires of the people who live and work on the farm.

"Programs live and grow, and their inhabitants -- the programmers -- need to work with that program the way the farmer works with the homestead" [Gab96]

Similarly, considering any software program as an analogy of a building, and programmers representing the occupants of software, the layer of change can be described in order to describe the activity of refining and reshaping software as the process of testing, debugging, extending, adapting and maintaining it through time.

2.The Architectural Development Pattern

Brief

The Architectural Development Pattern allows to consider a software program or system as a whole, to be designed and composed by different kinds of elements, which make it capable to be safely changed, adapted, and modified during its lifetime.

Examples

A Non-programming Example

The idea of layers of change in a building architecture arises from the form in which a building is planned and created, following the actual and future needs of its occupants. Based on an original description, Steward Brand proposes a general purpose "six S's" for the Layers of Change in building architecture [Brand94]:

- *Site. "This is the geographical setting, the urban location, and the legally defined lot, whose boundaries and context out lasts generations of ephemeral buildings".*
- *Structure. "The foundation and load--bearing elements are perilous and expensive to change, so people don't. These are the building. Structural life ranges from 30 to 300 years (but few buildings make it past 60, for other reasons)".*
- *Skin. "Exterior surfaces now change every 20 years or so, to keep up with fashion or technology, or for wholesale repair. Recent focus on energy costs has lead to re--engineered Skins that are air--tight and better--insulated".*
- *Services. "These are the working guts of a building: communications wiring, electrical wiring, plumbing, sprinkler system, HVAC (heat, ventilating, and air conditioning), and moving parts like elevators and escalators. They wear out or obsolesce every 7 to 15 years. Many buildings are demolished early if their outdated systems are too deeply embedded to replace easily".*
- *Space Plan. "The interior layout -- where walls, ceilings, floors, and doors go. Turbulent commercial space can change every 3 years or so; exceptionally quiet homes might wait 30 years".*
- *Stuff. "Chairs, desks, phones, pictures; kitchen appliances, lamps, hair brushes; all the things that twitch around daily to monthly. Furniture is called mobilia in Italian for a good reason".*

A Programming Example

Let us consider the case study exposed in the chapter 2 in [GoF94]: *"Designing a Document Editor"*, based on the design of Lexi, a text editing application developed by Calder. In this example the whole problem of designing such document editor is partitioned into seven subproblems [GoF94]:

- *Document Structure. "The choice of internal representation for the document affects nearly every aspect of Lexi's design. All editing, formatting, displaying, and textual analysis will require traversing the representation. The way we organize this information will impact the design of the rest of the application".*
- *Formatting. "How does Lexi actually arrange text and graphics into lines and columns? What objects are responsible for carrying out different formatting policies? How do these policies interact with the document's internal representation?"*
- *Embellishing the user interface. "Lexi's user interface includes scroll bars, borders, and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexi's user interface evolves. Hence it's important to be able to add and remove embellishments easily without affecting the rest of the application".*
- *Supporting multiple look-and-feel standards. "Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification".*
- *Supporting multiple windows system. "Different look-and-feel standards are usually implemented in different window systems. Lexi's design should be as independent of the window system as possible".*
- *User operations. "Users control Lexi through various user interfaces, including buttons and pull-down menus. The functionality behind these interfaces is scattered throughout the objects in the application. The challenge here is to provide a uniform mechanism both for accessing this scattered functionality and for undoing its effects".*
- *Spelling checking and hyphenation. "How does Lexi support analytical operations such as checking for misspelled words and determining hyphenation points? How can we minimize the number of classes we have to modify to add a new analytical operation?*

It can be noticed that these subproblems are related with the development order proposed for the design, as well as to the rate of change that each one may expose during the lifetime of the document editor. Certainly, there is a sense of logical order during their discussion. Furthermore, there seems to be a constant concern about the change, modification and adaptation capabilities that the document editor should reflect during its lifetime.

Context

In general, the Architectural Development pattern can be considered as a start out idea for the definition of the software architecture of a program or software system, aiming for maintainability as an important design attribute.

Problem

The problem is to develop software programs, which are able to evolve through time. In general, software programs are not built specifically to support certain evolution, in the form of change, adaptation, or modification to the original design and implementation. *"Time is the essence of the real design problem"* [Brand94].

Forces

Maintainability is a complex attribute. It is composed by several other attributes that act as forces to consider during the design:

- Change implies an internal ability to acquire new features or delete unwanted ones. It is related with adding a new functionality, enhancing an existing one, or repairing errors. It is important to answer how our program deals with improvement changes.

- Adaptation means the ability to cope with external changes. The software program should be capable to adapt to external changes in hardware and software technology, in order to remain competitive against other products in the market. The question here is what changes can be expected to happen in the technology environment.
- Modification is the ability to make changes quickly and cost effectively. It is important that a program is able not only to change and to adapt, but also to do it in a certain amount of time. The question here is how long does it take to change or to adapt an original version to an improved one.

Solution

First, it is important to notice that software programs are not static entities. They are the result of the activities performed by designers and programmers, in the form of layers of programmed components. Different parts of software programs change at different rates. Understanding this, the design, construction, maintenance and modification of software can be performed in a safer and more predictable way. However, a first difficulty seems to arise in how to recognise what parts of a software program represent each layer. In a software program, elements that compose a layer are not defined as clearly as in a building. Software Patterns can help to identify the layers of change in software, and also, to design and construct each one of the different layers needed to compose a software program.

Structure

The solution is presented as a layer-like structure. The use of this description is unfortunate because the term "layer" has a special meaning into the programming community. Layered software is composed of a number of different levels of abstraction. Each layer is allowed to communicate with the previous and next two ones [POSA96]. In contrast, the layer concept exposed here does not present precisely that behaviour. The layers are defined from a more natural point of view as stages of design and construction of software. Each layer is expected provide a base for the following one, but there is not explicit communication between them. Their relation is more approximate like a slippage between components. Even though it may cause some confusion, we decided to keep the description of layers to be consistent with the original idea. Figure 1 shows a brief of a notation used in [Benn97] for an architectural representation, relating components with the layers of change in software architecture. Their description is presented as the participants of this pattern.

Notice that Software Space Plan and Software Stuff do not form part of this architectural representation because both of them represent other levels of design detail.

Participants

- **Software Site.** The Software Site objective is to provide a stable base on which we construct software programs. This can be represented by the hardware elements of Computer Architecture and the software environment in which a software program will be developed. In general, a good part of the result of the software development and construction relays on these elements.

  During a software development, the elements that compose the Software Site should be analysed determined as the first elements that influence the design and implementation of software programs. *"Site is eternal"* [Brand94], at least during the lifetime of a software program.

- **Software Structure.** The Software Structure objective is to provide stability and support to the other subsequent layers. It can be represented by the basic organisation schema of a software program.

  Software Structure is the description of a software program as a set of defined subsystems, specifying their responsibilities, and including rules and guidelines for organising the relationships between them. Software Structure is concerned with the issues about partitioning a complex software system. The partition of software is necessary to cognitively deal with complexity. A big problem is divided into smaller subproblems that are possible to reason about, and perhaps perform some work on separately, at more comfortable cognitive level. Software Structure can be described in terms of architectural patterns:

  > *"An architectural pattern expresses a fundamental structural organisation schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organising the relationships between them"* [POSA96]

  Software Structure is properly the first design stage of a software program. In practice, software development consider the Software Site as provided or given, and initiates from the Software Structure design. Due to it supports and stabilises all other layers, any considerations and decisions taken during its design affect the other layers design and construction. *"Structure persists and dominates"* [Brand94].

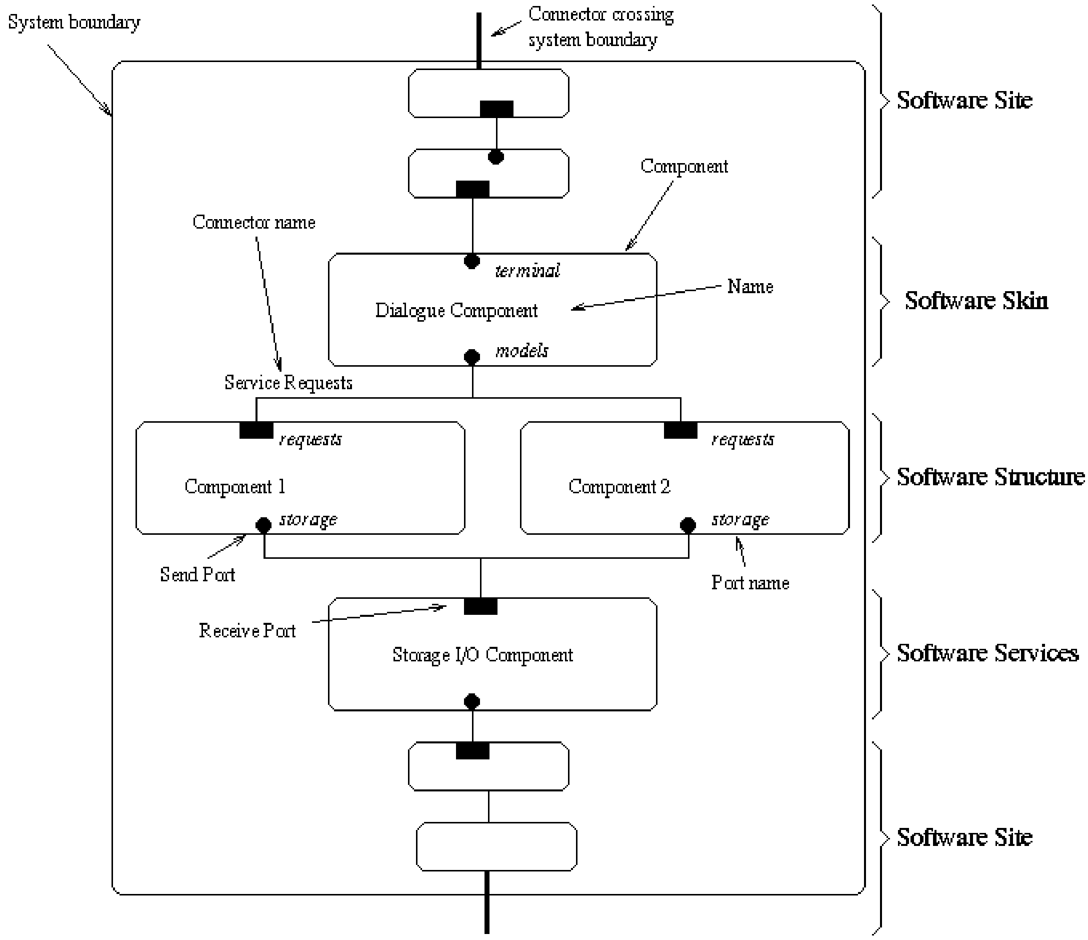

Figure 1. Notation for the architectural description of the Layers of Change in Software Architecture.

- **Software Skin.** The Software Skin objective is to give an appearance to a software program, exhibiting its functionality. In general, it is represented by all the elements that allow user interaction, mainly expressed by the use of graphical user interfaces, in which end users conceptualise how the software program works. Software Skin ranges from complete graphical environments to simple text screens. User friendly software programs try to enhance their Software Skin to make them attractive and understandable. The aim of Software Skin then is to support the usability of software, allowing users to learn about the software program and use it effectively.

  The most visible changes that can be performed on a software program are at the Software Skin layer. Just observe the graphic or programming environment of different users and programmers. Although they preserve the same elements, most of them represent a modified version of an original. Changes can be performed quickly and easily, according to necessity or taste. Also, it is noticeable that from one software release to another, developers commonly introduce changes in the user interface of a software program. The change may follow different causes, from newly added capabilities to aesthetics. These changes represent, in a more visible form, the improvement of the software as a product but occasionally, as with buildings, this is the only improvement of the software. *"Skin is mutable"* [Brand94].

- **Software Services.** The Software Services objective is to provide support for common activities during the use of a software program. Software Services are defined as those elements, which are part of the "working guts" of a software program. From a programming point of view, Software Services can be found in the form of all those pre-built standard components that provide common functionality like mathematical, input/output, and disk access, available as libraries of standard components from reliable suppliers or experienced programmers. Often, Software Services in a library are customised for a particular software design by the designer or programmer. For example, libraries of classes can be used effectively for customisation in Object-Oriented programming [Strous91].

  The correct use of Software Services can lead to more manageable, extensible and maintainable implementations. However, they are not universal standard programming tools or programming libraries. They depend closely on features and resources of the computer system where they are suppose to be used. Due to this, it would be unreasonable to expect them to be fully standard. Most Software Services can be considered standard on only a specific type of computer system [Strous91]. When computers change, Software Services have to evolve with them. *"Services obsolesce and wear out"* [Brand94].

- **Software Space plan.** The Software Space Plan objective is to organise the different partial tasks or activities performed by a software program. It represents the way in which the parts of a software program are organised. Following a particular paradigm or technology, data structures and functions are organised as abstractions in the form of software components and interfaces among them. Object-Oriented Programming, for example, presents organisations of classes that can be used as a layout of cooperative objects. This is the base for the concept of design patterns [POSA96,GoF94]:

  > *"The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"* [GoF94].
  >
  > *"A design pattern provides a scheme for refining the subsystems or components of a software system, or the relations between them. It describes a commonly - recurring structure of communicating components that solves a general design problem within a particular context"* [POSA96].

Design patterns can be applied in the design of the Software Space Plan by specifying detailed design aspects and the implementation requirements of a component. They can also be applied in refining and deciding on the basic module interfaces, following the reuse principle *"Program to an interface, not an implementation"* [GoF94]. Decoupling interface of implementation aims to simplify the reuse and reorganisation of the Software Space Plan.

As with the case of Space Plan in building architecture, Software Space Plan is the layer that often changes to cope with the needs and desires of occupants. This layer is where most software systems are designed to evolve in response to changes of existing requirements or the needs of new requirements. *"The space plan is the stage of the human comedy. New scene, new set"* [Brand94].

- **Software Stuff.** The Software Stuff objective is to represent the actual programming elements that perform processing or contain information: functions, procedures, data representations, data structures, etc. In an OO program, classes are defined with an interface controlling access to the data and functions, and an implementation that represents the coding of such data and functions. Idioms are the Software Pattern approach for describing Software Stuff.

  > *"An idiom is a low-level pattern specific to a program language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language"* [POSA96].

  Software Stuff is precisely the working material of the programmer. It is the layer of software that is continuously evolving and changing. *"Stuff just keeps moving"* [Brand94].

Dynamics

The description of the layers of change establishes not only the relations among different layers during the construction, but also dynamic relations between the different component layers during use. The stability of change in a building is based on the layers that change slowly over time. Listing the layers in order of the rate of change from slow to quick, we have: Software Site supports Software Structure, which stabilises Software Skin, allows connections of Software Services and brings together Software Space Plan, which is composed of Software Stuff. In Software, as in a building, the slow parts have the same objective: to provide stability to the system. Difficult problems arise when attempting to modify slow parts at a faster rate than it is allowed.

Influence also goes in the other direction: at times of radical change, the quickly changing layers influence the slow ones: slower layers tend to gradually integrate the trends of the quick layers within them.

In summary, the slow parts of a software system provide continuity and constraint, while the quick parts provide originality and change. *"The speedy components propose, and the slow dispose"* [Brand94].

Example Resolved

In this example, we develop and evaluate a software architecture for a Video Store Rental system, based on the Layers of Change for Software Architecture, in order to show how this pattern can be applied to a proposed problem. The problem selected for illustration is a variation on the Video Rental store, proposed by Norm Kerth [Kerth99], in which it is assumed a set of customer requirements, following the run-time and build-time requirements of the problem. Briefly, the problem states the need of a computer system by a video rental store to support its business. The customer requirements are basically presented in the form of scenarios to rent a product, to check-in a video, to establish and maintain a customer account, etc.

For the development of the example, two levels of detail to architecture development are considered: First, an initial architecture is developed, mainly exposing a proposal for Software Structure, Software Skin and Software Services. This initial architecture serves as a basis for preliminary feasibility evaluation, and as base for detailed design. Second, a refinement of the architecture is developed, using Software Space Plan and Software Stuff. Refinement can be performed when the detailed design is completed, as the output of later design studies. The basic constraint on the layers and components definitions is that they are fixed; once created, they remain as long as the software system executes. Considering this constraint and the functional requirement information as inputs, we can make our first approach to define the software architecture, trying always to design towards the big picture.

1. Specifying Software Site.

The Software Site is considered as a logical boundary of the system, representing the hardware architecture. Any element exterior to this boundary is considered an element of the surrounding system environment. By now, the ports are not physical ports, but eventually will be terminals for the users, and connections to peripherals that the system environment provide, and support the architecture. We start by drawing a box, and labelling it the Software Site. Ports to the exterior can be defined by inspection, looking at the list of potential users. A port for each kind of user seems a reasonable start. If several users are expected to share or use the same kind of terminal interface, the number of ports can be reduced. For this example, we will assume that the manager and administrator will use the same terminal, and that all clerks will use different terminals. Because the system may grow to other stores, the administrator may be connected to the system by a network. The manager uses either a local terminal or network connection. It is helpful to show which external users will be connected through each port.

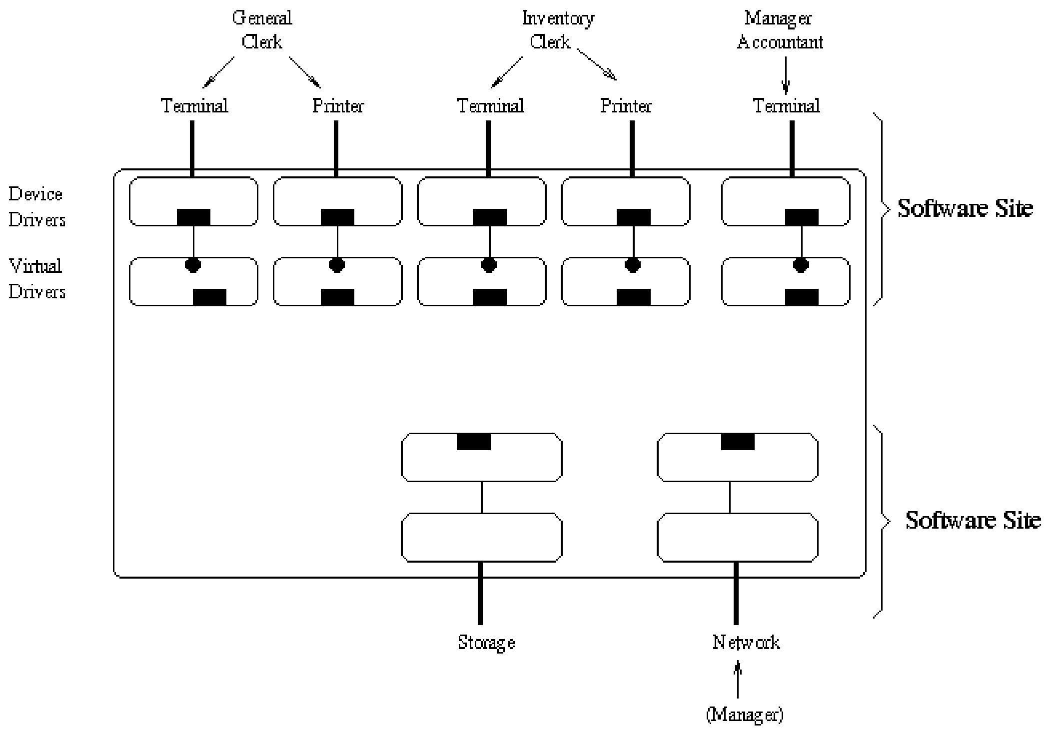

Figure 2. Software Site components

In general, aiming for independence from hardware, it is convenient that each port is associated with a device and a virtual device as interfaces between the environment and the Software Skin and Software Services, which at the same time cover the Software Structure as the central model of the application. The use of devices and virtual devices was chosen because most of the change cases can be supported with them, to provide for platform independence. Since the Software Site is represented here as a logical boundary, these low-level interface components are perhaps the easiest components to define. However, even if at this point Software Site is proposed as a physical hardware, the interface components are expected to facilitate portability, so it is worth while to consider them here. A good proposal is to define two for each port: one virtual device to isolate the application from the ports, and another device component to deal directly with the physical port. The Software Site now looks like Figure 2.

2. Defining Software Structure, Software Skin and Software Services

- **Software Structure.** The next major task is to define the Software Structure of the software system. At this stage, this means finding a high-level subdivision of the system into constituent components. These components should use semantics from the application domain, and they result as part of the decision of which Software Structure to use. However, eventhough we are aware of the different aspects, it is difficult to organize all of them into a workable structure. Some architectural patterns may be helpful here to perhaps obtain a useful organization previously used. Due to the characteristics of this problem, potential solutions could be the Blackboard pattern and the Layers pattern [POSA96]. As we are trying this example to be illustrative of the architectural approach to the present problem, a two layered Software Structure is simple enough to be proposed. Therefore, our initial approach is to divide the set of components between two groups: those which hold the state of the system, and those which perform functionality. We start following this order.

  First, let us provide a place to keep the system state. Data components seem a good place to allocate behaviours related to looking up specific components and creating new system state components. Basically, they provide entry points into the system state for the application. At this stage, without detailed static or behaviour definition, the best we can do is to guess at what the main entry points are. We can check those guesses with other similar cases, or perhaps some design patterns related with the chosen Software Structure. However, a complete evaluation will have to wait until results of the detailed behaviour design are available.

  In addition to defining data components, we should have some idea of their contents and responsibilities, as they will hold the system state, and can serve system state components to whomever asks for them. On the other side of the data components we should have a database interface. The design decision is that there should be no explicit database dependencies in the data components. They should be responsible only for maintaining their own state as long as the state can be determined with information available within the component itself. All of those opinions on the behaviours of the data components represent behaviour allocation definitions. They are not final specifications and will have to be confirmed by evaluation against external requirements that our system is supposed to support.

  Next, we define functional components to provide for functional extensions. This definition process is subject to the additional constraint that the functional components must provide the units of functional growth, change and configuration. The final Software Structure components obtained from the functional requirements are as follows:

- Process Queries
- Change Inventory
- Administer Customers
- Administer Cash Drawer
- Check Inventory
- Administer Users
- Inventory
- Members
- Transactions
- Cash Drawer
- Users
- Reports

When we developing the functionalities list, we have concentrated on handling extensions, so we can take this list as good enough for now. However, having the whole picture to look at makes a potential problem when defining connections between components in subsequent layers visible: how to arrange the connections between the future Software Skin dialogues and the functional components. Changing which functionalities are available to an individual user, and adding new functionalities in the future, are common change cases, and we aim to keep Software Structure as stable as possible during the system's lifetime. Those changes will take place on either side of the connections that relate the functionalities to the dialogues. The question is, do we need one connection, that is, one set of messages, for each kind of functional component, or can we use the same connection for all functional components?

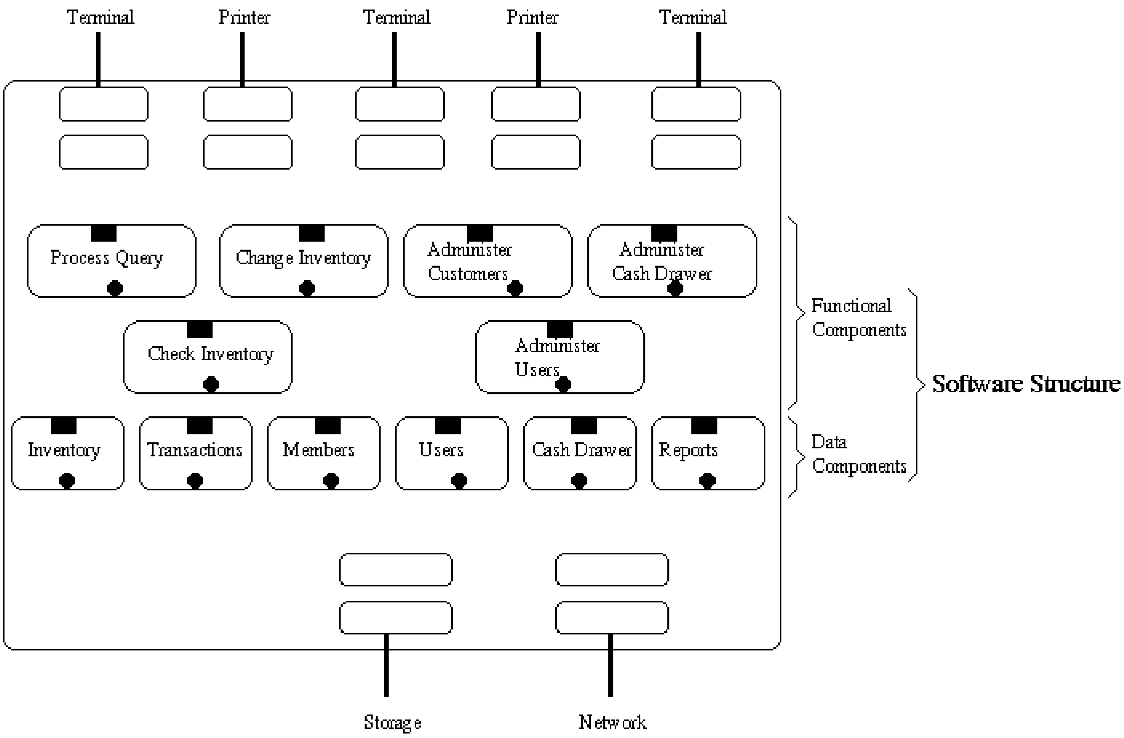

Figure 3. Software Structure components

The advantage of the one-connection approach is that adding a new functionality will not require changes to the external interfaces of the Software Skin dialogue components. Whether it requires changes to the internal interfaces of the dialogue components depends on where we allocate the responsibility for keeping the list of available actions for each functional component. The most flexible arrangement would be for the functional components to provide the command choices to the dialogues. This would make adding a new service a matter of registering the functionality with the dialogues that will offer it. This behaviour allocation scheme should be kept in mind when we define the Software Skin dialogues, and allocate the detailed behaviour to each one of them. This seems a good solution. However, we will return to this problem when defining the Software Skin of the system. The architectural description of the software system look now like Figure 3.

- **Software Skin.** Software Skin should provide the system for user independence. Dialogue components are responsible to carry out the mapping between events and requests from outside users and the functionalities and resources inside the system. Functionalities visible to the user may be provided by coordinating the outputs of several of the internal functional components.

  Initially, dialogue components may be associated with the user they serve or with the functionalities they coordinate. The user dialogues will include sales clerk, inventory clerk, manager, and accountant. The usage-sequence dialogues are check out, check in, inventory access, reporting, and administration.

  This set of dialogue components was arrived with some written ideas on our part of what kinds of responsibilities they will have. The user dialogues will be responsible for indicating which functionalities are available to their respective users, interpreting the users' requests and for invoking the appropriate sequence of control signals to have those requirements carried out. The operation-sequence dialogues will be responsible for actually managing the interaction between user and system while the user is checking out a video or adding

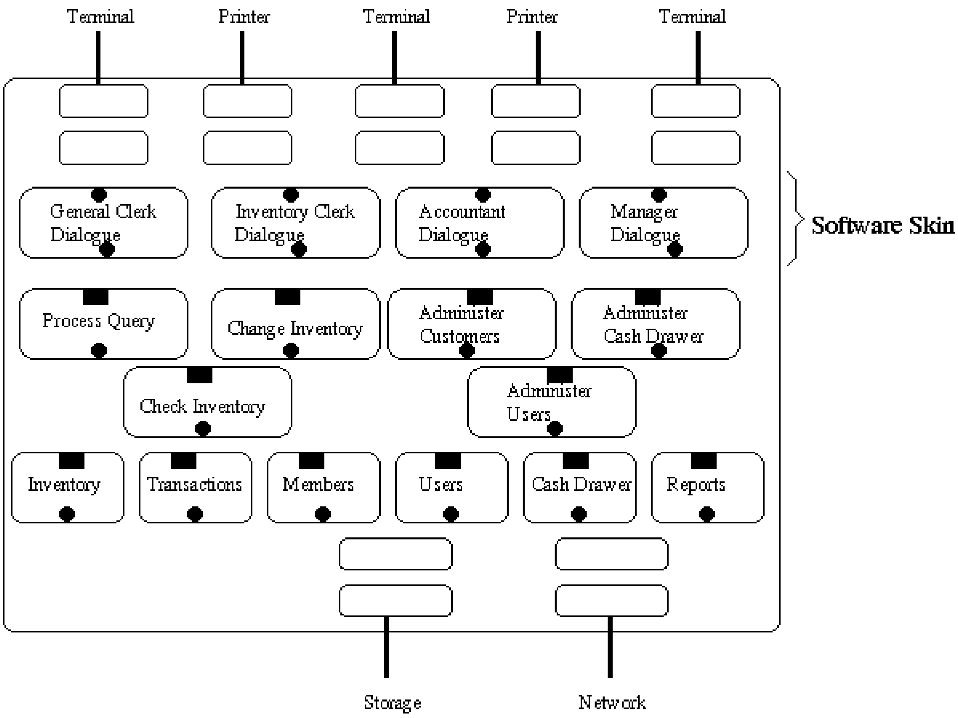

Figure 4. Software Skin dialogue components

to inventory. They will also be responsible for invoking the proper sequence of internal functionalities to provide the external services the user is requesting. This responsibility allocation will leave the user dialogues as “menu servers” for the users, which may be all right.

This meditation leads to the conclusion that the main unit of configuration should be the user, and perhaps the architecture should reflect that fact. Since it seems like the operation-sequence dialogue information will have anyway to be replicated for multiple users, the responsibilities originally allocated to the operation-sequence dialogues could be reallocated to user dialogues. This will leave us with a single level of dialogue components (Figure 4).

The responsibility description then is as follows: the user dialogues are responsible for all dialogue management with each kind of user. Each dialogue component will be configured with the appropriate controls for that user. The dialogue will invoke the appropriate sequence of operation and resource requests to provide the requested functionality. At this stage, the name and number of each port are just a guess, but they should correctly reflect the fact that the functionality allocated to the operation-dialogue components has not been reallocated to the user-dialogue components. Therefore, the Software Skin dialogue components obtained are as follows:

- General Clerk dialogue
- Inventory Clerk dialogue
- Accountant dialogue
- Manager dialogue

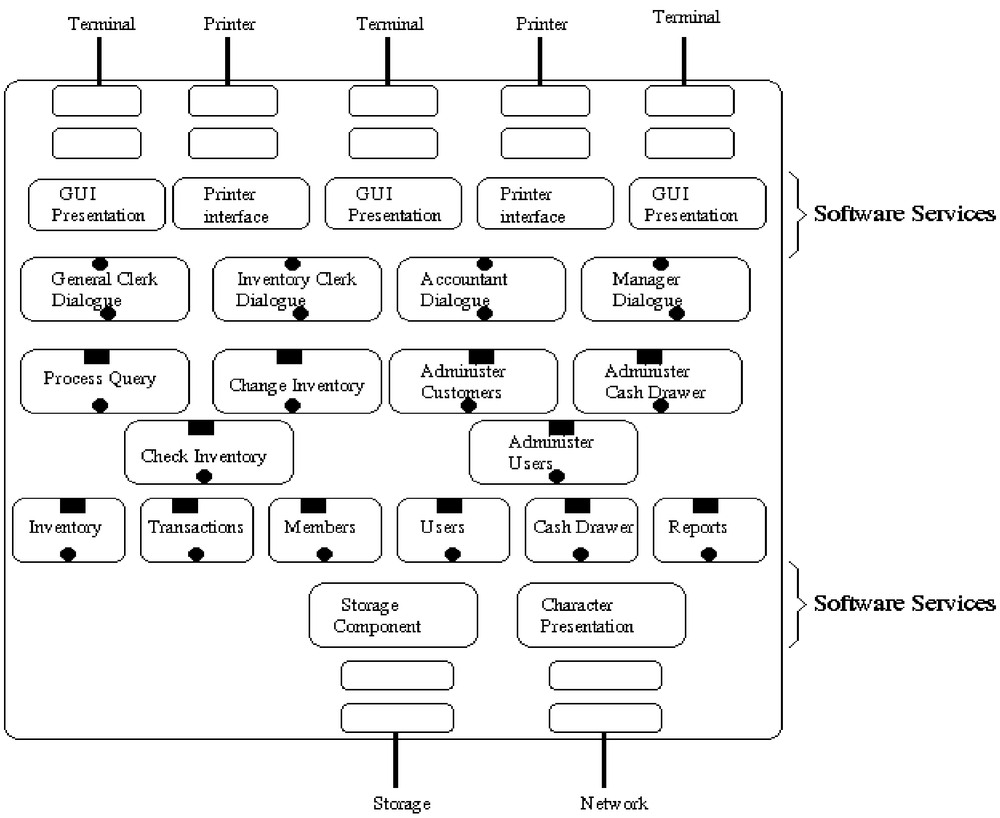

Figure 5. Adding Software Services

- **Software Services.** In general, Software Services are responsible for receiving and sending information to the system outside, so they are usually associated with a port and their behaviour is dependent primarily on the nature of the device connected to the port, or ports, they are serving. There should be at least one per port, unless two ports share the same information (for example if the connection is a network with several different kinds of users on the other end). In particular, Software Service components are appropriate for any interface that requires significant translation between the formats on the inside of the system and the formats on the outside. Printers, networks, and databases are examples of external systems that can use a separate Software Service (Figure 5).

  In order to provide for services independence in this system, we propose to define one Service component for each of the ports connected to peripherals. The network port, for instance, may have several kinds of devices connected to it (for example storage systems and remote users). For now, we will assume that the component needed for remote users is different from the component needed for local terminals, so we define a remote user component and a storage component to serve the network port as Software Service components. When the design is more stable, these component definitions can be changed with little effect on the neighbouring systems.

3. Organizing Connections

Connection definition and their content come from the interface views allocated to each component it connects. After the allocation decisions are made, we can derive the interfaces for the architecture components directly from the data components that were allocated to each component. Each message whose receiver becomes part of the interface between the components holding the sender and the receiver. The component collaborations that result can be viewed as lines indicating that there is some communication between components. We can populate those lines with actual messages.

The collection of messages may give us a view that we have not seen before. For instance, we have gone to some trouble to separate the Skin and Service components from the rest of the application. The motive was that they may represent possibilities of change, and be reusable within this application and even across multiple applications. A recommendation to keep message consistency is to choose a single set of messages, which should be mapped back to the defined component behaviour. The diagrams should be redone using the new message sets for the components.

After determining that the messages are sufficient and consistent, they should be organized into connections. Connections are as an important part of the architecture as the components. The connections are the basis for component reuse within a single application and across multiple applications, and it is important to define them carefully, to arrive at a consistent connection definition, combining the messages from other interfaces into one consistent superset of the current messages. From this information, it may be possible to find relations between components that have been modelled before, and are now documented as design patterns [POSA96, PLoP94, GoF94, PLoP95]. These can be used at this level for the design and implementation of all connectors in the description.

4. Allocating Software Space Plan and Software Stuff

The Software Space Plan is the detailed development of components, using the Software Stuff as the local data structures of each one of the components. Up to this point, architecture has been shaped primarily by the change cases. The user required behaviour has not been accounted for in any very explicit way. In the previous section, it was assumed that there was no detailed behaviour available, but several considerations could be taken. Here, that we should make that behaviour available. We should have a complete set of specifications which, when taken together, provide all of the behaviour and information required by users.The form of those specifications should be either collaboration responsibility specifications or interface views. Design patterns and Idioms can be helpful here for implementation of tasks at component internal level.

The main task here is to merge the architecture and behaviour to produce a system that successfully addresses both the user and development-sponsor requirements. The process requires a set of behaviour allocation decisions. Inputs are a list of behaviours observed in use cases, and the architecture components defined. The results of allocation decisions may be shown in two formats: each behaviour analysis may be followed by the names of the architecture components to which they are allocated; or architecture components may be followed by the behaviour analysis it contains.

5. Evaluation

At this point, we defined an initial set of components and their possible connections, completing a first pass at the definitions for the software architecture. A complete architecture diagram with all proposed connections is shown in Figure 6. A key aspect of the process was the continual reference to the growth and change issues. This method is appropriate because it is in the definition of the software architecture that the quality attributes, such as maintainability, extensibility, modularity, etc., get designed into the final product. By being aware of the change cases while we are defining components and allocating responsibilities, we increase the likelihood that the design is as modular as it needs to be. Being good designers, however, requires that we explicitly demonstrate that our design definitions meet the requirements. For that reason, we should carry out the evaluation described next. Simple evaluations, done early, are very helpful.

The inputs to the preliminary architecture included use cases and some change cases. The evaluations we can do at this point consist of qualitative walk-throughs of the use and change on the architecture.

- **Completeness.** As an initial check, the architecture diagram can be examined for general completeness. Some considerations that may be visible include whether there is sufficient hardware provided to handle the external users and peripherals, and whether all the connections that cross the boundary are accounted for. When there are multiple users, we should see that there is a representative of each of the major user types. In the diagram, the paths from external users through the dialogue components seems to be clear. Another aspect of completeness that can be easily evaluated is whether all that the system manipulates have been accounted for. Here, there may be some problems. For example, the existence of some components may be implied by the names of the components that are supposed to manipulate them. If those components do not appear in the set of components, it may be advisable to add them as data components. Now the question here is if these new data components need to be separated from the functional components that manipulate them. There are two criteria for separating a functional component from a data component. One is if there is more than one functionality that accesses the data and the other is if the functionality and data might change at different times. Evaluating on the first criteria requires a more detailed behaviour description than we have right now, so to be safe, let us keep the functionality components separate from the data components. Later, we can merge them if evaluation shows that they are tightly coupled.
- **Evaluating for User Case Requirements.** Evaluating the ability of the architecture to provide the use cases can be done by walking through the use cases. A recommendation is to do these walks quickly, because if

they are done in great detail, the effort is equivalent to doing a collaboration-irreparability behaviour model with the components as objects. While this is feasible, it reduces the independence the user requirements and the development sponsor requirement.

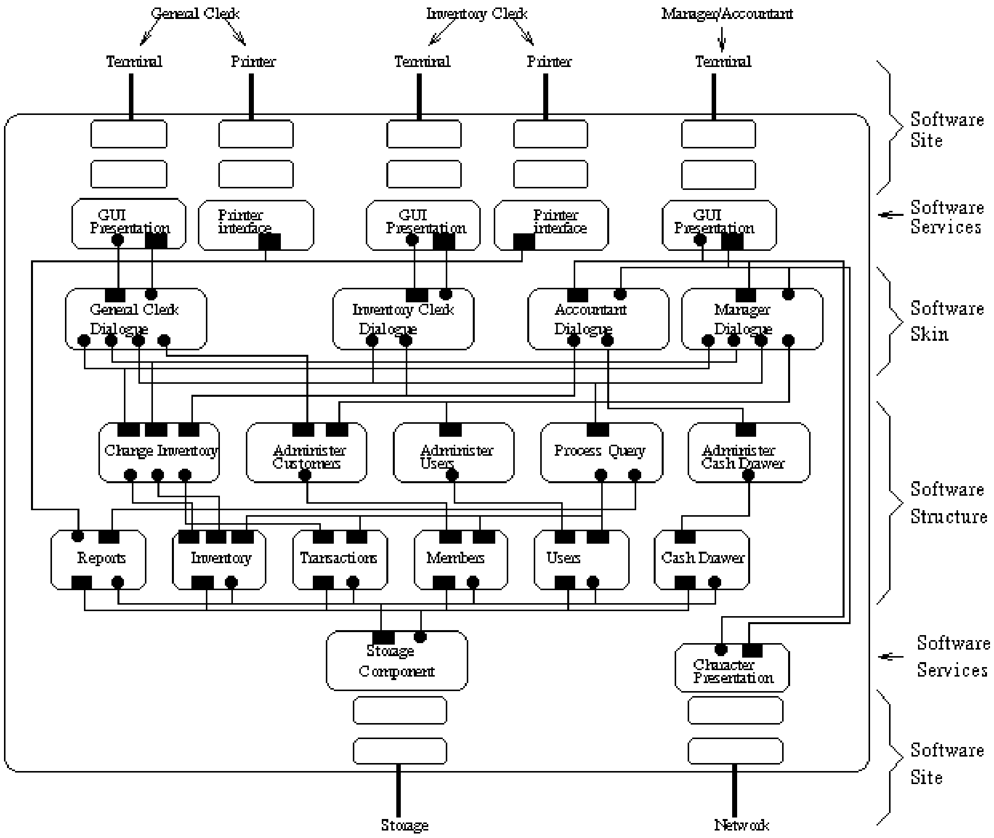

Figure 6. Example architectural description with proposed connections

The question is, for each use case, can we start with a user and see a path through the components that looks like it will provide the required behaviour? Here, "looks like" means that the subsystem names imply they can do what is needed.

Evaluating for Change. The three considerations that should be asked of each change case walk-through are:

1. If one or more components are impacted by the change case
2. If one or more connection interfaces result changed through the change case
3. If behaviours and information in the impacted components are not directly related to the change case

Change Computer Platforms

The evaluation is a change that may affect the whole architecture. However, because of the definitions of the components, none of the behaviours and none of the component interfaces appear to have any dependency on the platform or operating system. By itself, this implies that there should be no impact of changing the platform or the operating system on any of the main application components. To ensure that this is the case, we should specify that there can be no operating system calls in any of the application-level components. Operating system and computer platform dependencies should be limited to the device and virtual device components at the Software Site boundary. For components whose implementation uses operating-system tasks, the scheduling of the tasks is probably provided by the operating system. The application code inside the components should have little or no exposure to the scheduling functionality. If this constraint can be met, the system should support the Change Platform case quite well.

Change Database

Only the Storage Service component should be impacted. The Storage connection done not present messages which are dependent on the particular database used, so it should remain unchanged when the database is changed. Storage Service component should have no behaviours that are not connected with moving things between the database and the application. It can be conclude that this architecture adequately supports this change case.

Add capability for custom report definition

Defining reports at run-time requires that the user would be able to see all of the information that can be reported on. The subjects of the reports are expected not to change. To implement this change case, we propose to add a new functional component, called Define Reports. This component would have the needed access to all the data components.

First, let us consider whether there are any existing components that would be impacted by the addition of this new component. Without a behaviour definition for this use case, which would be a good idea to develop, it appears that defining reports would require a quite complex user dialogue. It would be expected that there should be a change to any dialogue component that offers report definition to its users. Also, it seems that there is already a repository for reports in the system. It would be very convenient if the form of the script for general reports were the same as the form for the custom reports. So the answer seems to be that only dialogue components will be impacted by the addition of the new functional component.

Second, let us consider if the existing interfaces hide the change case. Remember that the convention established in this design is to provide a separate connection on the dialogue components for each major operation. Following this convention would mean that adding a new capability would require a new connection for report definition on each dialogue component that offers the new capability.

Finally, Define Reports needs to manipulate the report scripts in the Reports component. The only connection available for Reports is Report Retrieval, which has messages to get titles and get report. These are not sufficient for creating and editing reports. As the aim is not to change data components or their connections when capabilities are added, it would be advisable to fix these connections to support this future requirement. We have two obvious choices for the fix: add more messages to the existing connection, or create a new connection. We could add more message due to there are not many new messages needed, but the idea of separating common operations from editing operations looks very appealing. So, the proposed solution will be to add a new connection called Edit Reports. As a first approach to message content, let us consider operations to get titles, get report, and add report. Now, it is necessary to return to the behaviour definition with these suggestions, and describe the interactions necessary to actually carry out the report definition.

The conclusion for this change case is that only the existing Reports component would need to be fixed to support the change. Additionally, when the change is made, it will require adding dialogue components and a new connection to one or more dialogue components. No changes should be necessary in any other component.

In summary, it can be noticed that in defining the architecture for a software system there are no right or wrong answers, only trade-offs. It may be noticed that the architecture and its evaluation in this example do not represent a perfect design, but they depict the line of thinking contained in this pattern. Our intention is not to show a perfect design, but to demonstrate a process for developing and evaluating designs, and for finding and fixing problems. There are a lot of problems in this example, some of which were discussed, a few fixed, and some which have not been found yet.

Consequences

Benefits

- When a software design project is started, a description based on the Architectural Development pattern allows to investigate design approaches at different levels, to uncover potential problems. The description allows also more detailed design studies, to get a better estimation on costs and risks. Finally, it also supports the production prototypes that may be designed and built to work out product details. An important benefit can be obtained when applying this pattern in case of large and/or complex software systems.
- While all software programs change with time, only some adapt and improve. The difference between an adaptable software program and one that is not adaptable is perhaps due to the relations between the Layers of Change. Identifying and respecting the rate of change of different layers in a software program allows the original designer and his/her successors to be able to navigate through it, to understand what changes to make, and to make them safely and correctly. Software Patterns can help in to obtain the expected "slippage" between layers, when used during the design and construction of a software program. The aim is to create software that is able to adapt and improve. *"Age plus adaptivity is what makes a building come to be loved. The building learns from its occupants, and they learn from it"* [Brand94].
- Other collateral benefits of the Architectural Development pattern are that its use enhances reusability by relating programming elements with each layer. It also improves integrability and changeability by associating each element with a context of design and use.

Liabilities

- Even though each layer does not interact with the others, it influences their development. If a lower layer is not well defined, an error may propagate during the development of the subsequent layers, influencing negatively on them.
- When change is to be performed at lower layers, especial care should be taken. Time and cost difficulties may easily arise for some changes involving lower layers, like portability and restructuring, which affect directly the Software Site and Software Structure.
- During design stages, a lot of information should be considered to specify each layer. Occasionally, this information may not be available or correct at design time.

Known uses

- The Solo Operating System [Brinch77] is a simple but useful single user operating system for the development and distribution of Pascal programs for the PDP 11/45 computer. This is the Software Site in which the system is constructed. The description of Solo continues by introducing its Software Structure that consists of a hierarchy of program layers, each of which controls a particular kind of computer resource, and a set of concurrent processes that use these resources. The user communicates with the system through a console. Since the development of this system is previous to GUI's creation, a very simple interface allows the user to edit, compile and store Pascal programs. The actual physical resources are controlled by a set of concurrent processes that allow the system to execute programs by operator's request, to feed the input processes with information stored in punched cards, to print lines sent by the output processes, etc. These concurrent processes are Solo's Software Services. During the description of each one of the layers composing the structure, The Software Space Plan for each one is introduced, describing each layer in terms of abstract data types as monitors and classes. Finally, Software Stuff is a description of the data structures and their implementation in Pascal [Brinch77].

- The Architecture Example for a video store system [Benn97] is a clear example of the use of the Architectural Development pattern. First, he starts proposing the Software Site by defining the system boundary, and representing the devices and external actors, which influence the development of the system. Then, he nominates the model, presentation and interface subsystems, representing in order, the Software Structure, the Software Skin and the Software Services of his system. At the next stage, he adds a list of services that define the model behaviour, such as process queries, change inventory, administer members administer cash drawer, check inventory, change users, define reports and process bills. These services are provided by one or more elements that are affected according to the kind of service. So, each model component is made responsible for a group of shared actions. These define the Software Space Plan. Finally, he proposes an incremental implementation of each component, which represents the Software Stuff of the video store system [Benn97].
- CORBA (Common Object Request Broker Architecture) is an example of the Architectural Development pattern [OMG90]. CORBA represents an application of OO concepts to distributed systems. An Object Request Broker (ORB) allows objects to publish their interfaces and allows client programs to locate them anywhere on a computer network, requesting services for these remote objects. The Software Site of CORBA is then this network of distributed workstation systems. A client computer program makes requests of services provided by another object implementation. Both client and object implementation use the same ORB interface. This Software Structure used is similar to the Broker architectural pattern [POSA96]. CORBA applications may or may not present a Software Skin. In the first case, they use GUI's to communicate with end users, such in the case when integrating CORBA and, for instance, Java applets. Software Services are represented by the ORB code that performs connection management and all the low-level details of interacting with a remote object. The CORBA components use several patterns to achieve their function, representing the Software Space Plan. Especially, the Bridge pattern [GoF94] and the Proxy pattern [POSA96, GoF94] are used at this level. Finally, the implementation of CORBA services, and the actual data structures can be seen as the CORBA Software Stuff [OMG90].

Related Patterns

The Architectural Development pattern is expected to relate in general with all pattern categories: Architectural patterns [POSA96], Design patterns [POSA96, PLoP94, PLoP95, GoF94] and Idioms [POSA96], trying to coordinate their use in a more ordered and consistent form.

3. Summary

Why is it often simpler to understand architectural properties in building architecture than in software architecture? It is probably because a building is the most commonly human constructed system that we know. A building is a stage for human life. We usually spend all our lives in buildings, and know about their advantages and liabilities, whereas only recently the idea of “inhabiting” software has been considered by the Pattern Community.

Surely, this is not the only description analogy that can be obtained between building and software architecture. Our interpretation of the Layers of Change is not unique. Interpretations depending on other paradigms, techniques and applications can also be proposed. However, as our work is closely related to the area of Software Architecture, Software Patterns, and Object-Oriented Programming, this analogy seems to fulfil our requirements and expectations. Our aim is that the way we use analogy to obtain this interpretation for software design and construction would be useful for others to propose their own interpretations [OR99].

4. Acknowledgements

We are specially grateful to our shepherd for EuroPLoP’99, Norm Kerth, for his comments, suggestions and critics, providing invaluable insights and profound reflection about the contents to the improvement of this work, and his permission to include the problem in its entirety. The work presented in this paper is part of an ongoing research by Jorge L. Ortega Arjona in the Department of Computer Science, University College London.

5. References

[APL77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[TWoB79] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[Benn97] Douglas Bennett. *Designing Hard Software. The Essential Tasks.* Manning Publications Co. 1997.

[Brand94] Steward Brand. *How Buildings Learn. What happens after they're built.* Phoenix Illustrated, Orion Books Ltd., 1994.

[Brinch77] Per Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice-Hall Series in Automatic Computation, 1977.

[POSA96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, Ltd., 1996.

[PLoP94] James O. Coplien and Douglas C. Schmidt, eds. *Pattern Languages of Program Design,* Addison Wesley, Reading, Mass., May 1995.

[Gab96] Richard Gabriel. *Patterns of Software: Tales from the Software Community.* Oxford University Press, 1996.

[GoF94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Systems.* Addison-Wesley, Reading, MA, 1994.

[Kerth99] Norman L. Kerth, Customer Requirements for Video Store Rental. Elite Systems. P.O. Box 82907, Portland, OR 97282-0907. `nkerth@acm.org`

[OMG90] Object Management Group and X/Open. *The Common Object Request Broker: Architecture and Specification.* OMG Document No. 91.12.1 (revision 1.1). 1990.

[OR99] Jorge L. Ortega-Arjona and Graham Roberts. *The Layers of Change in Software Architecture.* First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas, USA. February, 1999.

[Strous91] Bjarne Stroustrup. *The C++ Programming Language.* Second Edition. Addison-Wesley, 1991.

[PLoP95] John M. Vlissides, Norman L. Kerth, and James O. Coplien, eds. *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996.