Managing the software generation process

By Ir J.A.J. van Leunen

Retired physicist & software researcher

Location: Asten, the Netherlands

Website: www.scitech.nl

Communicate your comments to info at that site.

ABSTRACT

The current software generation process is rotten. This paper analyses why that is the case and what can be done about it.

Contents

Software complexity

It is no secret that the generation of complex software poses great problems for its producers. The cost is growing exponentially with their size and the time from conception to finalization grows likewise. The resulting products are fragile and force the vendors to reserve sufficient resources to cope with future warranty and damage claims. Buyers are aware of this situation but without reasonable alternative they are ready to live with the situation. The source of the misery is the complexity of the software and this complexity is mainly due to the relational complexity of its constituents. A radical modular approach as is applied in hardware system generation would cure the problem, but that requires a completely different way of software generation and software marketing.

Introduction

First the factors that hamper efficient system generation are treated independent of the application area. Then the solutions for eliminating these factors are given. Next the differences between the hardware area and the software area are shown in an historic view. Finally a possible improvement of the software case is sketched.

Managing complexity

Breaking level

Managing simple projects hardly ever poses problems. However, a situation in which complexity surpasses the boundary where a quick view no longer reveals potential problems requires special methods. These measures compensate or cure the lack of overview. The level of the boundary depends on the number of items involved in the process and on the nature of the relations between these items.

Measure of complexity

The number of potential relations between the items involved in the process explains a close to quadratic growth of potential complexity with the number of items involved. Between N items exist N×(N-1) potential relations. Usually only a small percentage of the potential relations are truly relevant relations. Dynamically relevant relations are the potential carriers of communication and control signals. They carry the activity and determine the capabilities of the considered system. It takes expertise knowledge to decide whether a potential relation is dynamically relevant. Gaining this expertise takes time and other resources. This explains why all potential relations have a direct impact on manageability. For that reason, the number of potential relations may act as a rough measure of potential complexity. Similarly the number of dynamically relevant relations may act as a rough measure of the actual complexity of the system. More precise measures will also consider the type of the relations. The type of the relation determines how that relation must be treated.

Procedures such as modularization of the system and categorization and grouping of the interrelations into interfaces significantly reduce the actual complexity of system design and creation. Each interface represents a well defined group of dynamically relevant relations. Well known interfaces contribute *significantly to the reduction of complexity. They reduce a set of interrelations to a single relation. Modules can be assembled into systems by connecting them via compatible interfaces. Both the modules and the interfaces that couple these modules are of crucial importance for managing the complexity of system generation.*

Extreme complexity

Very high degrees of complexity may introduce secondary effects that impair manageability far more severely than can be explained by the number of potential relations between the items involved in the system generation process. This occurs when it becomes humanly impossible to properly specify the activity of all dynamically relevant relations.

The inability to specify the product, implicates the inability to test it and as a consequence it implicates the inability to guarantee the proper functionality of the system. The implications of the lack of resources that are required to cope with complexity and the inability to specify the situation in sufficient detail can easily raise costs in an exponentially increasing way. Apart from causing unacceptably growing costs, the system generation process yields fragile results. The resulting product may even endanger the environment where it is applied. This requires reserving resources to insure resistance against future claims.

The modular approach

Modularization

The reasons why modularization significantly improves manageability of the generation process are manifold. For example, it may be possible to delegate the design or the creation of modules to other parties. Potential reuse of existing modules or their design is another important reason. However, the most important reason for applying modularity is the fact that proper encapsulation of the modules and the use of well known interfaces significantly reduce the number of dynamically relevant relations.

A simple example may explain this. A monolithic system consisting of 1000 items contains 999.000 potential relations. Its relational complexity can be characterized by this number. A comparable modular system that consists of ten modules contains far less potential relations. Let the modules be coupled by well known interfaces and let part of the interfaces be similar. Not every module connects to every other module. Let the largest module contain 200 items and let the total number of interfaces between any pair of modules be less than 5. The largest module has a potential relational complexity of 39.800. The complexity of the other modules is less. Thus the relational complexity met by the module designers is less than 40.000 and for most modules the relational complexity is less than 10.000. Between modules the interfaces take the role of the relations that are the internal members of these interfaces. The system designer is confronted with a relational complexity that is less than 100. The benefits of the reuse of interfaces and the advantages of the possible reuse of modules should also be considered. Thus compared to the monolithic case there is an increase in manageability of several orders of magnitude. Modularization of larger systems may offer benefits that are much higher. Diminishing relational *complexity translates directly in lower man costs and in shorter time to realization. Further it has a very healthy effect on the robustness and reliability of the end product.*

Modular system design

The system designer gets the strongest benefit from the modularization. Modularization simplifies system assembly significantly. This opens the possibility to automate the system integration process.

Modularization reaches its highest effectiveness when the design and creation process enables the assembly of modules out of other modules. In this way the microelectronics industry reaches the exponential growth of the capabilities of integrated components that is known as Moore's law.

Interfaces

In the design of a system the introduction of an interface increments the number of potential relations. However, because the interface encapsulates a series of dynamically relevant relations, the total relational complexity will decrease. The new relation will only play a dynamic role when the corresponding modules are coupled or decoupled. This coupling can be done at system assembly time or during the operation of the system. At instances where no coupling or decoupling is performed the new relation acts as a static relation. It relays the communication and control signals to the dynamically relevant relations that are members of the interface. In the count for complexity a well known interface replaces the combined contributions of its members. In that view, it can be considered as a single dynamically relevant object.

Dynamically relevant relations are carriers of information or control signals. Depending on the direction of the control signal the corresponding interface member belongs to the require part of the interface or to the provide part of the interface. In the first case it acts as the sender of control signals. The require part of the interface contains members that belong to the current client module. If the interface member acts as the receiver of control signals, then the interface member belongs to the provide part of the interface. In return the interface member causes the module to deliver corresponding services. The provide part of the interface belongs to the module that acts as the current server. In order to become active the require part of the interface of the client module must be connected to the provide part of the interface of the module that acts as the server.

A module may act as a server at one instance and it may act as a client at other instances. In each of its roles it will use the appropriate provide or require interface parts. Multitasking modules may provide parallel actions.

In the assembly the coupling of the require interface part and a corresponding provide interface part may be stationary or it may be temporally. The provide interface part of an interface may serve one or more *other interfaces. The service may be presented in parallel or in sequential order. The specifications of the provide interface part must at least cover the requirements of each of its customers. With respect to its potential capabilities, the provide interface part may offer more than is requested by a coupled require interface part. The specification of the provide interface part must be in accordance with the specification of the require interface part, but this only holds for the part that covers the services that the require interface part may demand.*

In many cases the trigger of a provide interface member by a connected require interface member will not only result in an action of the server module. It may also cause the return of a response via the same connection. The response can be used for synchronization purposes and it may contain requested information.

In general an interface may contain both a provide part and a require part and the partition may change dynamically. It is difficult to understand and handle such mixed interfaces. When manageability is strived for, then mixed interfaces must be avoided. An exception exists when the communication requires a handshaking process. Preferably pure interfaces should be used. A pure interface contains either a require part or a provide part but not both. In the simplest case the specification of a require interface closely matches the specification of the corresponding provide interface.

Proper modules

Proper modules are properly encapsulated. A proper module hides its internals. Securing the intelligent property that went in its design is one of the reasons for this strict measure. Preventing unwanted access to the module is another reason. Proper modules can only be accessed through publicly known and well specified provide interfaces. A module is a part of an actual system or it is targeted as a part of one or more possible future systems. Proper modules take care that each access through an interface keeps the functional integrity of the module intact. An exception may be that the module signals to its environment that it is no longer in a valid state. The environment may then decide to ignore the module in future actions or it may reset the module to a valid state.

A proper module must be able to perform one or more actions. These actions may be controlled via one or more of its provide interfaces. Purely static objects are never considered as proper modules.

Properties and actions

Each proper module has a set of properties that together describe its status. Besides of that, each proper module provides a series of actions. Each module interface offers indirect access in order to control the members of a well defined and ordered subset of these actions. The properties cannot be accessed directly. However, a given action may enable the reading of the value of a property or it may enable the direct or indirect setting of one or more properties.

Costs of modularization

Modularization has its price. The design and generation of modules and the organization of compatible interfaces is relatively expensive. Only extensive reuse of modules may render modularization economic. Reuse of modules and the availability of compatible well known interfaces between modules may significantly improve the manageability of the design and creation of complex systems. However, reuse implicates standardization and it asks for actions that promote availability, accessibility and diversity. These requirements are best provided by a healthy and lively modules market and media that publish the specification of the characteristics of available modules and interfaces. An open market may ensure a healthy price to quality ratio. It also stimulates the continuous improvement of the quality of the modules that become available. Preparing modules for an open market requires the hiding of the intellectual property that is invested in the design and creation of the module. On the other hand the specification of provide interfaces must be publicly known. Promoting other uses of the provide interfaces and the require interfaces that are applied in a given module will in its turn promote the use of that module. It will increase the chance that other modules will become compatible with the considered module.

Abuse

Modularization can also be abused. Wrong access to a module may rupture its integrity. In that case the module is no longer trustworthy. A proper modularization technology must prevent improper access to modules. It means that access that bypasses the official interfaces of the module must be prevented. Clients of a module may be systems or other modules. During its actions a module may run through a sequence of states. A client of a module must only access the module while the module is in a state that is known to be save for this access. Properly created modules will then take care that their integrity will not be impaired. If the state of a module is not known, then the client may decide to reset the module to a save known state.

Abuse of modularity is stimulated by the misuse of the terms 'module' and 'component'. It often occurs that a system part is called 'component' or 'module' while it is far from properly encapsulated. Such system parts are not designed to preserve their integrity. People that do not have sufficient expertise may fall into this trap and may think that by assembling such improper components a similar reduction of complexity can be achieved as can be achieved with proper modules.

Modularization success cases

The success of modularization is widely demonstrated in the design and generation of hardware. Electronic appliances, autos, buildings, clothes, in fact most assembled products are not affordable without the fact that they are constructed from components. Many of the constituting components are themselves assembled from components. More important, the price, quality, diversity and availability of these components depend strongly on the corresponding lively components markets. The beneficial effects of the open market depend strongly on trustable specification of the characteristics of the components and on media that report on availability and quality of these products.

Even nature relies on modularization. Most living creatures contain organs and are constituted from multiple cells. Human communities use modularity in the hierarchical structure of their organizations. This *is best shown in a town hall or a post office where dedicated counters belonging to corresponding departments offer publicly known services to their customers.*

The application of modularization in the software industry is far from a great success. Proper software modules exist, but their application is sparse. The current software development tools do not support the assembly of systems from modules. The software components rely on the support that is offered by the operating system that embeds these components. Most software components are designed to operate as singles in a larger non-modular environment. Generally, these modules do not couple with other modules. Currently the software industry does not offer a technology that enables the construction of modules out of other modules.

Requirements for success

When applied properly, modularization may significantly improve the system design and creation process. Keywords are the standardization, the diversity and the availability of modules and interfaces and the ease of the system integration process. The existence of a lively and effective modules market is also a very important aspect. System integration may be automated, but this requires the proper tuning of component specification, the system design tools and the matching components market. The technology must enable the construction of modules out of simpler modules. Using these preconditions the microelectronics industry provides very complex and tremendously capable integrated circuits.

With a proper automated design and assembly organization in place the modular system creation time will shrink to a small fraction of the time required by the manual non-modular equivalent. Where manual design and assembly of a complex monolithic target requires a genius as the system architect, a creative human operator may burn far less resources and achieve a similar or even better result by using an appropriate automated modular approach. Automation of the system design and creation process puts high demands on trustworthy and machine readable specifications of modules and interfaces.

Difficulties posed by modularization

The requirements posed by modularization are also the reasons why modularization is never a straightforward solution.

Diversity

The requirement of a high degree of diversity is in direct conflict with the requirement of sufficient standardization. An interface has both static and dynamic aspects. Dynamic requirements may ask for different interfaces that have similar static characteristics but different dynamic behavior. Environmental requirements may ask for specially adapted interfaces. Interfaces may be replaced by other interfaces that have a wider scope or a better performance. Similar considerations hold for modules.

In order to increase market profits, to simplify component discovery and to ease system integration the diversity of similar interfaces must be kept within sensible bounds. The same holds for modules.

Compatibility

In order to enable successful assembly, the selected modules must be mutually compatible. This translates to the requirement that the interfaces that couple the modules must be compatible. Provide interfaces must cover the demands of the coupled require interfaces. The requirements include both static and dynamic characteristics.

Real time behavior of modules may require measures that prevent or cure deadlock and race conditions. The design tools must enable the installation of these measures. Other measures must prevent that the system runs out of essential resources. The modules must be designed to support these measures. When all relevant data of the constituting modules are known, then the system design tools can help the system designer to implement sufficient resources and to take the appropriate measures.

Platforms

Components may be designed for different application areas. For example software may be designed for desktop purposes, for servers or for embedding in electronic appliances. In each of these cases there exists a choice of hardware platforms. Electronic hardware platforms require adapted software components and will certainly influence the dynamical characteristics of the interfaces of the software components. Mechanical modules may target automotive systems, avionics, nautical systems, stationary instruments or other mechanical systems. Each application area and supporting platform may require its own range of modules and interfaces. Each application area requests an adapted components market and an adapted system assembly technology.

Hiding intellectual property

In some application areas the hiding of the intellectual property that went into the design and the creation of modules is provided by their physical form or by market conditions such as a patent system. However, some application areas currently lack sufficient means to hide the design of the components. Without proper IP hiding a component's creator can never make profit in an open components market. In the past, this fact has certainly prevented that the software industry developed a healthy and lively software components market. This does not say that it is impossible to generate an effective IP hiding system for software modules.

Availability

Availability is assured when several suppliers exist for popular modules. An easily accessible publication organization must promote and enable the discovery and the selection of existing modules.

Specification

The specification must be accurate and complete. The specification must contain sufficient details such that the system integrator can determine how the considered module can be assembled with other modules into a target system. Automated assembly asks for a machine readable and therefore well standardized specification format. This requires a dedicated XML format. The format can be defined in an XSD document. For humans, an XML document is not easily readable. The XML document can be made readable for humans via one or more XSL documents. The specification of the statical characteristics of an *interface is well established. Currently there exists much less support for standardized specification of the dynamical characteristics of interfaces.*

Hardware versus software

History

The hardware industry booked far more success with the application of modularization than the software industry. Partly the volatile nature of software is responsible for this fact. However, the differences in the evolution of the corresponding design and creation technologies had more influence on the success of modularization.

Long before the birth of electronic computers, modularization took its position in hardware industry. Computer hardware became affordable through far reaching application of modularization. The early computer programmers used machine code as the language to communicate with the computers. Soon the burden of inputting all these codes separately was eased by an assembly compiler that translated assembly terms into corresponding machine code sentences. Program parts could become reusable routines. Libraries of these routines became products that could be applied in different programming projects. The next step was the introduction of the third generation languages. These tools offered a better readable and much more flexible coding of the functionality that the programmers had to write. Powerful compilers translated the source code and combined it with the precompiled library members that were called by the written program.

Basic architecture trends

Up to so far this was no more than easing the process of producing machine code. The growing complexity of the programs demanded software development tools that enable a better overview of the architecture of the design. At this point two trends developed.

Functional analysis

The first trend, phrased 'structural analysis' created a split between the handling of properties and the handling of the actions that influence these properties. The methodology collected properties in 'data stores', actions in 'processes', data messages in 'data flows' and control messages in 'control flows'. The graphical representation of the result of the analysis was called a 'data flow diagram'. In advance, the approach proved very successful. It led to the introduction of several important software development items such as, routine libraries, file systems, communication systems and data bases. Most third generation programming languages and the early software development tools supported the 'structural analysis' approach.

Abstract data types

The second trend promoted the modular approach. It used 'abstract data types' introduced by David Parnas as its modules. In design phase the 'abstract data type' acted as an individual. It was well *encapsulated and could only be accessed through one or more interfaces. In the seventies of the last century the complexity of most software projects did not enforce a modular approach. For that reason this modular design methodology was not well supported by programming languages and by corresponding modular software development tools.*

Object orientation

In a later phase the complexity of the software design increased such that a more modular approach became necessary. Instead of taking the proper modular approach of the 'abstract data type' the main software development turned to object orientation. Here the objects resemble 'abstract data types', but the objects are not properly encapsulated. Access via interfaces is possible, but the client of the object may also access the actions of the objects more directly. More severely, often the internal properties of the object can be altered directly by external actors. The possibility to inherit functionality from an object with a simpler design was given much more attention. The result was the development of libraries of classes of objects with a deep inheritance hierarchy.

Currently, object orientation is well supported by software languages and software development tools. Pity enough, current object oriented software development tools do not promote the use of popular interfaces.

Object orientation has some severe drawbacks. Without sufficient precautions, classes taken from different class libraries cannot be combined in programs. A class library with a deep inheritance hierarchy may become obsolete when its top classes contain services that are no longer up to date with current technology.

Current software components

The software industry also came with more proper software modules. Examples are Microsoft's COM components and the Java Beans. COM components are supported by some operating systems and Java Beans are supported by the Java virtual machine.

The support for COM in software languages and in software development tools is small. The design of the architecture of the COM skeleton prevents trustworthy memory garbage collection management in cases where the module can be removed dynamically. COM is supported on some embedded systems that use UNIX or an operating system that supports POSIX.

Both Java Beans and COM components are not designed to construct components from components and need the support of an operating system or a virtual machine.

There exists a small open market for these software components. Most of them target desktop applications.

State of affairs

At this moment the software industry does not apply modularization in a serious way. There exists no theoretical reason why modularization in software system generation can not be as successful as the current modularization in hardware system generation currently is. However, effective modular software generation asks for a completely different way of software generation than is accomplished by the present software development industry.

Implementing proper modularization will offer chances to parties that are now excluded by the power of companies that control software development tools and software development processes. With the appropriate services in place, everybody who has access to a software component development environment can produce products that fill a market need. Future institutions that support software component development and component based system assembly will help the component developer in marketing the created components. In that case the current powers in the software industry will endanger losing market control. It is to be expected that they will battle to stay in control.

Coupling the market and the design and creation of software modules and interfaces

Standardization and marketing

Modularization asks for a dedicated and powerful standardization of specifications, interfaces and coupling procedures. A globally accessible service must support the distribution of the public documents. For example, dedicated web based repositories may contain standardized and categorized specification documents that can be discovered by an appropriate search mechanism. The development tools must be able to access the specification contents contained in these documents. Another globally accessible service must support the gathering, the sale and the delivery of the corresponding components. Both services must cooperate.

The tools and the services must intimately interact to enable the quick and efficient design of interfaces, components and target systems. At the same time the services must ensure that the intelligent property that is invested in the uploaded components keeps hidden from the public world. It must also be guaranteed that the component designers will get their rightful fee. It is very difficult to organize a properly controllable pay per copy of the components binary. It is suggested that the customers pay per project for each used binary.

Designing and generating components

The component designer collects the required interfaces from web based or local repositories or he designs one or more new interfaces. Then he designs and creates one or more components. He must test these thoroughly. When ready he uses the components for local system design or he packs one or more *components into a package and sends this together with the appropriate documents to the institute that will market his products. The institute checks the contributions and after a positive conclusion the institute puts the binaries and documents in its banks. The institute will put the documents in the appropriate repositories where they become publicly accessible. Users of the components may buy the components from the institute. The institute will ensure the payment of the developer that has put the product in the bank.*

Versions and diversity

Versions and diversity of components both impede and support the manageability of the system integration process. Therefore the number of versions must be limited. Diversity of components must be made manageable by reducing the number of supported platforms and by limiting the number of supported environments. Development and creation of close copies of existing components must be avoided. Breaking these rules can easily destroy the advantages of modular system design.

Hiding intelligent property

Hiding intelligent property that is invested in the design of the component is one of the most difficult points of software component technology. It can be arranged by power: excluding customers from future membership when they offense the 'rules'. Or it can be ensured by a combination of encryption en recompilation supported by a hardware decryption. Every project gets its own encryption key. It must be ensured that a system designer can still use components that he himself has designed and created.

Automating system integration

The system integrator starts with collecting the required application components and with creating the necessary connection and scheduling scripts. The components are put in packages and a project document defines the target. Because of the fact that at the start of the system integration practically all relevant data are known, the system integration tool can automatically add a dedicated supporting operating system that includes automatic memory garbage collection. The retrieved component specifications suffice to enable the construction of skeleton systems. After linking, these skeleton components can already be tested. However, the 'empty' components do not produce much activity. During system development the skeleton components can be replaced step by step by fully operational binaries.

Publishing

Publications related to modularization comprise specifications, market promotion media and product quality comparison reports. The internal code of components is normally hidden. If the institution that designed the component wants this, it is possible to make this code public as part of the component specification.

A fully fledged software components industry

Sketch

There exists no theoretical reason why proper modularization cannot be achieved for software as it is done for hardware. The realization of some aspects will be easier while the achievement of other aspects will be harder. It is easier to send software products over internet. It is easy to search the document repositories *of the component shops for interesting components and compatible interfaces. Using XML it becomes feasible to automate the design and creation process that makes use of these web based repositories, which contain machine readable specification documents that describe components and interfaces. A local file based equivalent of such a repository may store retrievals and new designs and serve both the system designer and the components developer. The repositories contain a search machine that looks for categorization terms that classify the specification documents for specific application areas. New designs can be uploaded to a central service that will check the information and store it in the worldwide accessible repositories. A webservice that acts as a dedicated web based shop may offer the corresponding modules. In the background of the webservice, binary banks will hold the binaries of the modules. The webservice will use a dedicated money bank to support the financial part of its activity. Via the webservice the component designers may upload their results to the central institution that will then market their products. Component development tools and system assembly tools interact with the repositories and the webservices to implement an integrated design, assembly and marketing environment.*

The demo

This is a very sketchy view of a possible implementation of an integrated software components creation and marketing system. In order to investigate the feasibility of this sketchy picture a demonstration system is built that contains working versions of all important constituents.

The demonstration system supports:

- *Embedded software and desktop software*[1]
- *Provide interfaces*
- *Require interfaces*[2]
- *Memory mapped hardware interfaces*
- *Streaming interfaces*
- *Notification interfaces*[3]
- *Package*[4] *of a coherent set of components.*
- *Components*[5] *that consist of simpler components.*
- *Automatic creation of the supporting operating system from dedicated modules*[6]
- *Stepwise system build-up from a mix of skeleton components, partially functional components and fully functional components*
- *Automatic memory management*
- *System modes*[7]

[1]In embedded software the generated system interacts directly with the hardware. The system assembly tool adds the HAL.

[2]Require interfaces are implemented as placeholders for special types that represent a reference to a provide interface.

[3]Notification interfaces accept hardware triggers.

[4]A package is a library of a coherent set of components. A component supplier will preferably deliver his products in the form of packages. A system designer will save his subsystems in the form of packages.

[5]A composed component is a dedicated package accompanied by a dedicated (fixed) connection scheme and a dedicated (fixed) scheduling scheme.

[6]In embedded software the system integration tool generates operating system modules in C++ source code. In desktop software the system design tool generates a layer that interacts with the virtual machine. This layer is generated in source code that corresponds with that virtual machine (C# or java).

[7]System modes are controlled by connection schemes and scheduling schemes. Dynamic removal or creation of modules should be restricted to the instances where the system mode changes. Memory management is also restricted to these instances.

A standard RTOS schedules threads by stopping and starting routines. In a component based environment the real time scheduler must stop, reset and start modules. Eventually the modules must be reconnected according to the currently valid connection scheme.

The demonstration system consists of the following components:

- *An example of a web based repository*
    - *This repository exists of a hierarchy of directories that contain*

        - *XML documents, which contain structured specifications. Each document contains a series of categorization tags.*
        - *XSD documents, which define the structure of the specifications*
        - *XSL documents, which help convert XML documents into humanly readable documents*
    - *The repository has a hierarchical structure. Components and interfaces are assembled in separate directories.*
    - *The repository is publicly accessible. Using the XSL files the XML documents are humanly readable via a modern web browser.*
    - *The repository contains a search machine that uses the attached category tags to find corresponding documents.*
- *An example of a local file based repository*
    - *This repository exists of a hierarchy of directories and has the same structure as the web based repository. This includes the search capability.*
    - *The local repository contains a larger variety of documents than the web based repository.*
    - *It acts as a local store for information that is retrieved from one or more web based repositories.*
    - *It acts as a local store for documents that are prepared to be send to a general institute that may put these documents on a web based repository.*
    - *The XML documents specify:*
        - *Component*
        - *Interface*
            - *Require interface*
            - *SW/SW*
            - *HW/SW*
            - *Streaming*
            - *Notification*
        - *Types*
            - *Plain type*

            - *Enum type*
            - *Interface type*
            - *Sequence type*
            - *Structure type*
        - *Package description*
        - *Connection scheme*
        - *Scheduling scheme*
        - *State chart*
        - *Project description*

- *An example of a webservice that may act as the representative of a central institute. This institution serves the community that creates or uses software components. Components may appear as packages of simpler components.*
    - *The institute owns a local repository that contains all specifications of interfaces that exist in the domain of the webservice.*
    - *The institute owns a binary database that holds the binaries of all available software components.*
    - *The institute owns a local repository that contains all specifications of software components that exist in the domain of the webservice.*
    - *The webservice uses the binary databases and the local repositories to automatically serve the customers of the institute. Customers have no direct access to these stores.*
    - *The webservice helps partners of the central institute to distribute documents to their specialized web based repositories.*
    - *The webservice helps customers in buying software components and retrieving the corresponding binaries from the binary bank*
    - *The webservice helps software component developers to upload the binaries and corresponding specifications of their products.*
    - *The central institute takes care that the software component developers get paid for products that are downloaded via the webservice.*
- *A repository browser tool*

    - *The tool helps with searching local or web based repositories for existing interfaces and components. Selected documents can be transferred from the web based repository to the local repository.*

- *An interface and component design tool*
    - *The tool helps with specifying new interfaces. This includes:*
        - *Software-software interfaces*
        - *Software-hardware interfaces*
        - *Streaming interfaces*
        - *Notification interfaces*
    - *The tool helps in specifying other design documents that go into the repositories.*
    - *The tool helps with searching local or web based repositories for existing interfaces.*
    - *The tool helps designing and creating the skeleton of a software component*
    - *The tool helps with filling the skeleton with dedicated code*
    - *The 'internal' code is normally hidden. However, it is possible to make this code public with the rest of the specification.*
- *A system assembly tool*
    - *The tool helps with searching local or web based repositories for existing software components. It can retrieve the corresponding binaries from web based or local binary banks.*
    - *The tool can work with components that are still in skeleton form.*
    - *The tool can check whether components can fit together.*
    - *The tool assembles selected components and adds a dedicated component based operating system.*

Some hard rules must be obeyed.

- *All components and all interfaces have a globally unique identifier.*
- *Any binary and any specification document that is uploaded to the central institute and that is accepted by this institute must never be changed or removed.*

- *New versions of an item are related to the previous version via a relation document that is attached to the specification document.*
- *The number of new versions of an item must not surpass 4.*
- *Close copies of items, that are not new versions, will not be accepted.*