CMS TN 96-114 SW
21 August 1996

Object Oriented Development "The New Design Problem" [*]

Massimo Marino [†]
CERN, CMS/CMC
Geneva, Switzerland[‡]

New techniques often involve innovative approaches and looking at the problem they address from a different perspective: OO software development is not an exception. Evolutionary development and the adoption of OO will be of heavy impact on the software construction. The technological shift from classical development and structured methods to evolutionary development and object-oriented methods is certainly not easy. We must given it the time and the means in terms of structures, training, staff, and support for all to come effective.

CMS has joined several R&D projects to test if and how Object Orientation can be applied to its software. We share here our considerations on OO development and the understanding obtained through practical experiences within the CMS object oriented activities and the RD41 (Moose) project.

1. Introduction

Software development is a very demanding activity. Designing is not an exact science: different designers can produce different models of the same problem, no matter which development method is adopted. Furthermore, designing system for the HEP environment is inherently complex, essentially due to the life-span required to the software to survive and the geographically dispersed developer teams. To most physicists there is no way-out: software will be complex to develop, hard to manage and maintain, but it is to be there anyway. HEP applications do not escape the need to have a disciplined approach to software development in order to achieve its final aim: a reliable computer program that performs its tasks properly.

The development of any software systems should be carried out using a development method. At the base of every development method, in order to manage the complexity of software, we find a *divide and conquer* principle: the system is split into understandable and consistent chunks of information that form the units to be developed.

[*]http://ecpmoose.cern.ch/~marino/html/OOD.html
[†]marino@ecpmoose.cern.ch
[‡]ECP Division, CH 1211 Geneva 23, Switzerland

Today, the existing methods for software development can basically be divided into two categories: functions and data (procedural) methods versus object oriented methods. Comparing to traditional methods, building software applications within the object paradigm may result in a design that is clearer, simpler to manage, more robust with respect to changes.

2. Divide and Conquer

Traditional structured analysis - structured design (SA/SD) methods, applied to a variety of software development areas, as well as to the HEP environment, treat functions and data as separate entities. Such an approach often has lead to problems, especially during maintenance, since functions and data structures turn out to be quite sensitive to changes and additions. A system developed using a functions/data method often becomes difficult to maintain and evolve.

A major problem is that, in general, all functions must know the storage or the inner representation of data. Different data often have different formats, which means that we need to add conditional statements to identify the data type to take proper actions. To change a data structure we must then modify all the functions related to that structure. The system easily becomes unstable: any slight change will, in general, generate major consequences.

Object Oriented methods do not separate functions and data but view them as an integrated chunk. The OO approach is to understand the system by developing a model that is based on concepts and objects directly found in the problem domain. The resulting objects contain data and behaviour that describe the entire system functionalities. The concepts and objects in the problem domain have an higher chance to be stable than data structures, hence the overall architecture of the system will settle faster.

The problem domain provokes changes during the software life cycle. Within OO paradigm they will have a local impact on the software by affecting few objects only. Because of the very foundations of the object oriented paradigm, changes in internal details do not spread in the system architecture.

2.1. *SA/SD - Focus on Functions and Data*

Structured analysis and structured design, with all the models most physicists are used to[a], approaches system decomposition through a set of processes, the data they manage, and their mutual dependencies. SA/SD focuses upon and exposes data structures and the implementation details of the processes that manipulate them. These are exactly the parts in a complex software environment that are the most subject to changes for improvement.

Indeed most of the effort is spent first in trying to achieve stable data structures, then the hierarchical tree of processes that mutually exchange them. Unfortunately, once defined, processes are so tightly dependent on data structures that they become

[a]data flow diagrams, algorithmic decomposition, flow charts, top-down module hierarchical trees...

difficult to modify, let alone to improve. The knowledge of data structures is so deeply interwoven in the fabric of the system to discourage any evolution: even major improvements are abandoned. In other terms, the modules that express the relevant abstractions depend upon the modules that contain the irrelevant details. The abstractions get affected when changes are made to the details!

In SA/SD, the top-down tree of the structured program expresses the chain of dependencies from the more abstract modules at the top (closer to the problem domain) to the more detailed ones at the bottom, the purely implementation concerned (solution domain) modules. If we attempt to reuse one of those abstractions we must carry along all the details that those abstractions depend upon. It is not an exaggeration to say that every experiment in HEP has faced these and similar problems during its life time.

2.2. *OO - Focus on Interfaces*

OO uses a completely different perspective: data and processes are hidden within objects that have interfaces and responsibilities. An application is a set of objects that collaborate with each other to fulfil these responsibilities. In many respects this is a modelling improvement: we can describe applications in terms of **interfaces** instead of data and processes, that is, instead of implementation structures. The interfaces become software entities in all respects and depend neither on the software that uses them, nor in the software which implements them! Interfaces are to be seen as screens behind which we can hide many different data and processes: they represent stereotypes relieving us from the burden of knowing their exact implementation, hence leading to more flexible and robust designs.

What we aim to with OO is that dependencies are no more on the data model and the processes that manipulate it (as arising from data and functional decomposition) but on the interfaces only: given a client and a server, no part of the client has a dependency upon the server internals; instead the client depends on the interface that hides the server itself. Having the dependencies only on the interfaces also implies that changes do not propagate into other parts of the application. The changes will be contained within a particular client or server and will be irrelevant to other software chunks elsewhere in the system.

More importantly, the interface does not depend on the server either. The server is also made to depend on the interface: it is said the OO makes a **dependency inversion**. We can reuse both the server and the client in other contexts and even separately, with all the economical and quality gains that may result from this. Only the interfaces need to be implemented, the re-users being legitimated to implement those interfaces by any means they choose.

OO development is the technique through which the realisation of one of the thoughest goal of software engineering is within reach: components that realize the Open-Closed Principle ([Meyer, 1988]). One can enhance components without disturbing their existing capabilities. Components are *open* in that they may be extended without affecting the fabric of the system or its architecture. At the same time, they are *closed* in that they perform as black boxes accessed through their interfaces. Extensions are made by adding new software rather than by modifying existing one. Clients need not to know the internals of the objects which react only via the published services. When most of the modules conform to the Open-Closed Principle, the working code is not exposed to breakage. This creates a significant amount of isolation between features and allows for much easier maintenance.

Behaviour properties must be in foremost consideration: we adopt the Liskov Substitution Principle (LSP) for proper arrangement of classes in inheritance hierarchies ([Liskov, 1988]):

> If for each object *o1* of type **S** there is another object *o2* of type **T** such that for all programs **P** defined in terms of **T**, the behaviour of **P** is unchanged when *o1* is substituted for *o2*, then **S** is a subtype of **T**.

In other terms, derived classes must be usable through the base class interface without the need for the user to know the difference.

This rule is a logical extension of the Open-Closed Principle. Consider a function **F** that uses type **T**. Given **S** a subtype of **T**, **F** should be able to use objects of type **S** without knowing it. Any subtype of **T** should be substitutable as an argument of **F**. If this is not true, then **F** must have test statements to determine which of the various subtypes it is using. And this breaks the Open-Closed Principle.

Not applying the Liskov Principle can lead to programming language equivalent of homonyms: types having the same signature but different semantics - subtyping driven more by implementation than by design. If aiming at reuse Meyers suggests *implementation hierarchy* ([Meyers, 1992]) which is simple containment of an object of one class inside another. Implementation hierarchy states that we reuse the functionality of the contained object. We may use delegation patterns, e.g., MEDIATOR, STRATEGY, TEMPLATE, etc., ([Gamma et al.], 1994) where some operations are forwarded to matching objects from other classes. Another option is the use of private inheritance where no subtyping assumptions is involved.

When accessed via their interfaces, objects may be used in various manners, or in different collaborations, as made explicit in the use cases. Clients only need to conform to what the objects expect: the programming by *contract* as expounded in [Meyer, 1988]. Note that there is a strong relationship between the LSP and the concept of design by contract.

The Open-Closed Principle is the core of many of the claims made for OO design. It is when this principle is applied that a system is more maintainable, reusable and robust. LSP is important to all applications that conform to the Open-Closed Principle. It is only when derived types are completely substitutable for their base types that functions that use those base types can be reused with impunity, and the derived types can be changed with impunity ([Martin, 1995]).

Needless to say, all this is extremely difficult, if not impossible, to achieve with SA/SD.

2.3. *And Objects will behave...*

In procedural techniques the software system is naked, its intimate, more vulnerable parts exposed: it only asks for being raped with fury by uncaring personnel. In OO, processes and data structures are hidden, subordinate to the objects, their interfaces and their responsibilities. Objects may be shielded by caring developers against intrusion from the outside.

Designers are not required to throw away all they know about design and start from scratch, but they must learn the OO technique, apply a completely different perspective, and acquire a different mind set to be integrated with the experience and the talent. And objects will behave...

3. The Use Case Approach

Today we see use cases, a concept first formalised by Ivar Jacobson in the 1987 OOPSLA conference ([ACM, 1987], [Jacobson, 1987]), being incorporated into several object-oriented methods, e.g., Objectory ([Jacobson et al.,1992]), Fusion ([Coleman et al., 1994]), the Booch method ([Booch, 1994]), the so called Unified Method ([Booch and Rumbaugh, 1995]), Syntropy ([Cook and Daniels, 1994]), and many others.

Use cases, or *mechanisms* in Booch, are a way to express the objectives of the software system. In [Jacobson et al., 1995] use cases are defined as "...a sequence of transactions in a system whose task is to yield a measurable value...";.

We like to define use cases as the *behavioural chunks* of the system to be.

A valid approach is to examine system requirements with the intention to identify candidate objects and establish interactions and interrelationships among them. In the context of such an approach, we adopt use cases as a conceptual description of the requirements.

We identify three primary motivations for use case creation:

- gaining an understanding of the problem,
- capturing an understanding of the proposed solution,
- identifying candidate objects.

Each use case is a description of the system in terms of how the user will see it, and in terms of the delivered measurable values to the user ([Jacobson, 1992]), as well as a system mechanism, "...a structure whereby objects collaborate to provide some behaviour that satisfy a requirement of the problem..." ([Booch, 1994]).

Use cases, as they are most commonly described and used, are rather functional in nature. This, by itself, is not really a problem. Many software practitioners have a good deal of experience in functional decomposition approaches to system building. Use cases present them with an opportunity to continue their functional view of software development.

However, we must be careful: use cases, and in this respect they substantially differ from functions, are not to be treated in isolation. In the ultimate end, they describe one of the possible object collaborations within the system. In most cases the same object participates in several use cases. That is, to have a chance of accurately design any given object, we have to account for all its appearances in the invented use cases.

While functions can be treated and developed in isolation by different teams and then integrated[a], use cases must be seen and managed in their globality. In this respect we are less prone to this functional creepism with an architecture-driven process ([Booch, 1996]):

- Specify the system's desired behaviour through a collection of scenarios,
- Create, then validate, an architecture that exploits the common patterns found in these scenarios,
- Evolve that architecture, making mid-course corrections as necessary to adapt to new requirements as they are uncovered.

Booch refers to these activities as the *analysis*, *design* and *evolution* of the object-oriented life cycle. The key point here, easily missed, is the "common patterns". The architecture, or set of categories of related classes, is the one that supports the entire set of explicited use cases, and the others to arise in the future.

A sure recipe for failure from the OO point of view is:

1. Determine the main (highest level of detail) functional capabilities of the system to be,
2. Write a high-level use case for each high-level functional capability,
3. Assign each high-level use case to a separate team for further elaboration (partitioning the tasks),
4. Eventually, based on progressively more detail being added to the use cases, each team would implement their particular partition of the system,
5. Integrate the efforts of each team into the final product.

This procedure suffers from a functional decomposition "front end" and an object-oriented "back end", from missing the system architecture, and from a delayed integration. We end up with a lot of duplication of effort due to the lack of a dedicated activity to identify those objects that are common to two or more partitions, and with scattered parts of many same objects across more than one functional partition. It is easy to find each part of the system having a different implementation of the same object: this results in a significant amount of re-design and re-coding.

We should avoid the temptation to use this functional view of the system as a basis for the creation of an object-oriented architecture for that same system. Objects and functions do not map directly to each other, and the architecture of

[a]The (in)famous *big bang integration* of traditional development techniques.

an object-oriented system is significantly different from the architecture of a functionally decomposed system. Recall the primary motivations for use case creation mentioned before, and that all the foreseen use cases contribute, in parallel, to the invention of the supporting architecture.

4. Object Interaction Diagrams

In many notations use cases are represented by interaction diagrams, schematically shown in Fig. 1, where the sequence of message flows progresses over the objects participating in the use case. In the diagram time flows from top to bottom and the event and method passing is supposed to take no time. Interaction diagrams trace the execution of a scenario.

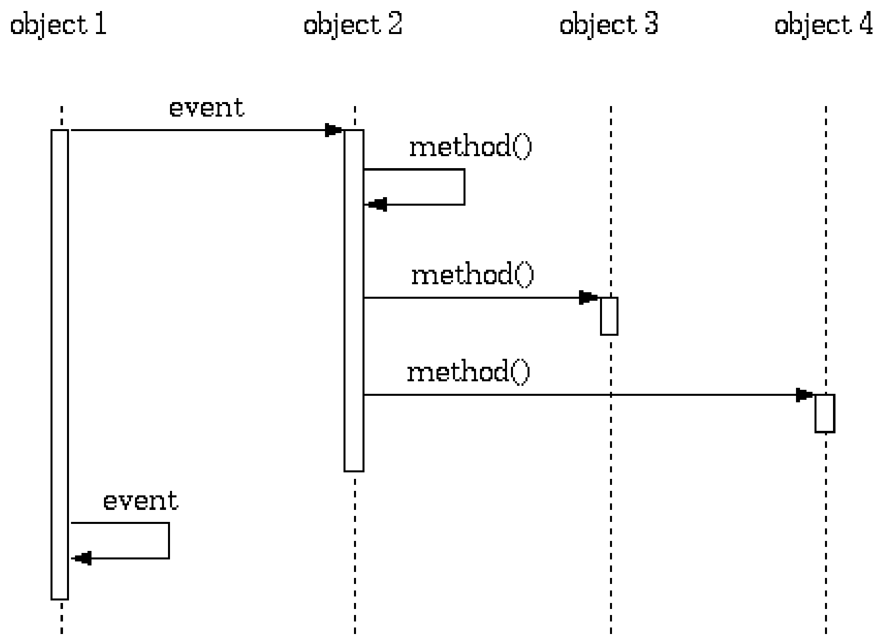

Fig. 1. Object interaction diagram .

In [Jacobson et al., 1992] interaction diagrams are said to make evident the decentralisation of the responsibility in the object collaboration. Figures 2 and 3 clearly show the extreme structures of the use case: the so called **fork** diagram and **stair** diagram.

4.1. *Fork diagram*

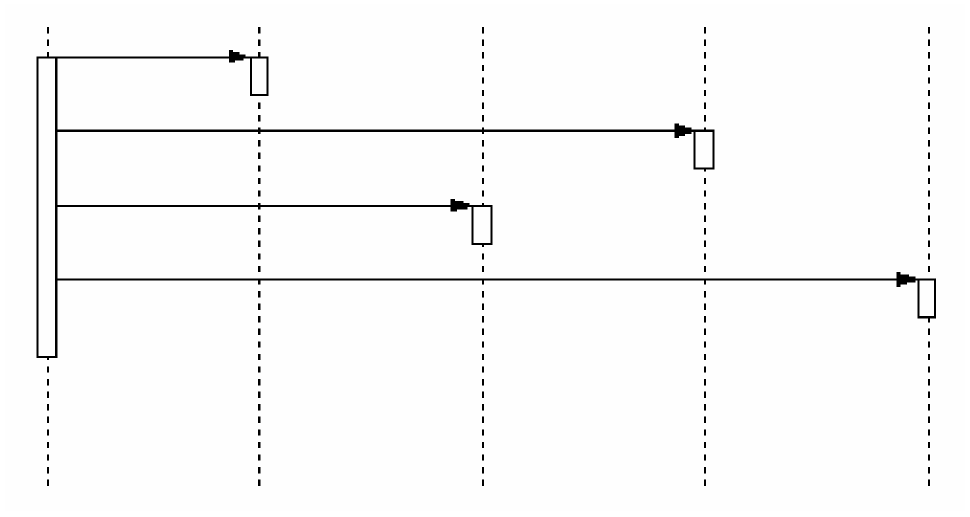

Fig. 2. Fork diagram .

A fork diagram arises in the presence of a central control. Pathological combinations of functional architectures and object-oriented ones are often encouraged by some of today's most popular methods, e.g., Objectory's so called "control objects", Booch's "manager objects", Meyer's "command objects", and the countless examples of classes that encapsulate functions only, i.e., the "stateless classes". The *controller* carries great responsibility and often becomes more and more complex during the software life cycle. The "intelligence" of this partition of the system is highly localised in the controller who also requires a lot of visibility: it has to know the presence of many other objects that alone are incapable of 'intelligent' behaviour. Indeed the resemblance with a main function and the related subroutines is striking.

If most of the system diagrams look like fork diagram, as in Fig. 2, an alarm bell should ring. This is especially true if the "called" objects (again the resemblance with subroutines...) are visible to the whole system, perform only under external control, and lack of mutual collaborations. Most probably the designers disguised data and/or function repositories or were just rewriting a functional/data architecture, maybe a legacy FORTRAN solution to their problem.

The fork diagram is absolutely valid if the use case describes the behaviour of a composite object. In this case the composite delegates its mandates to the private components, and no other object will be responsible for those parts. The use case shows then one of the *modus operandi* of the composite object and constitutes a detailed description of its internal mechanisms.

Some claim that there is nothing wrong with fork diagrams: they make explicit the situation where the developing team keeps the door opened to changes in the order of the operations managed by controller objects. Personally we saw fork diagrams mainly coming out at the beginning of our OO activities and fading away with the experience. In general it is not easy to introduce changes in the system operations with controller-based use cases, and becomes hard if the controlled objects are shared by many controllers. On the other hand the sequence of the operations is under total control of the manager object, but we do not look into object-oriented techniques to achieve this, do we?

If they cannot be said to be a sure proof of bad design, fork diagrams certainly make it easier to develop functional structures and should be carefully validated and justified.

4.2. *Stair diagram*

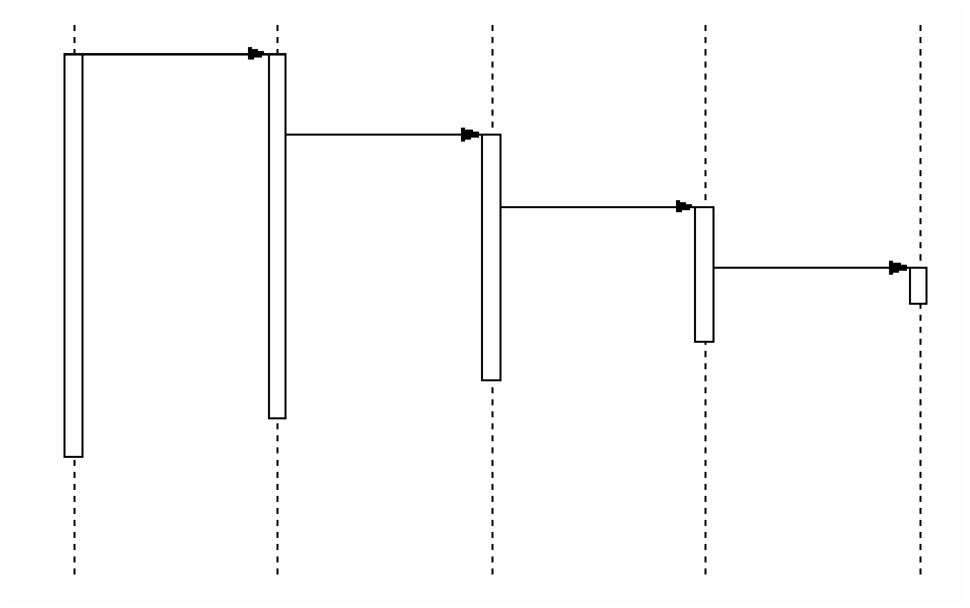

Fig. 3. Stair diagram .

The other extreme is represented by the stair diagram (Fig. 3). This structure clearly shows the delegation of responsibility. Each object knows only few of the other participating objects and we have no 'Do This - Do That' controller. The use case initiator object role is limited to triggering the delegation cascade. Each object has its own task and knows which other object can help to fulfil it.

Some subparts of these collaborations could easily find their place in other use cases as well and constitute collaboration chunks to achieve more complex behaviours. Here the responsibility is evenly distributed among the objects and there is no need of a distributed visibility of all participants.

The message here is not to avoid fork diagrams at all costs and strive for stair ones only. The important thing is to be able to recognise the structures, know what they do imply, and what they cost. In practice the structures are a mixture of the two extremes where some object has more "control" than other less independent objects. The lack of a controller and the decentralisation of responsibility is beneficial to the flexibility of the architecture, and the localisation of the information. It also allows to reuse some collaboration patterns of the system in different contexts.

The following pictures (Figures 4-5) show stair diagrams extracted from one of the CMS prototypes, a local method pattern recognition ([Innocente and Marino, 1995], [Bos et al., 1995]) as practical examples of application of the Open-Closed Principle. The primary mechanisms behind the principle are abstraction and polymorphism. These are implemented by inheritance in statically typed languages like C++. It is by using inheritance that we can create LSP derived classes that conform to the abstract polymorphic interfaces defined by pure virtual functions in abstract base classes.

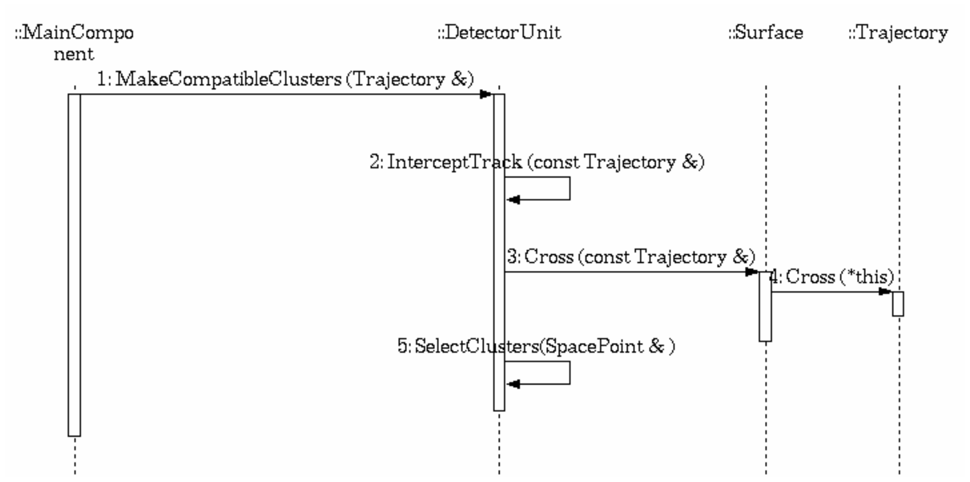
::MainComponent
::DetectorUnit
::Surface
::Trajectory
1: MakeCompatibleClusters (Trajectory &)
2: InterceptTrack (const Trajectory &)
3: Cross (const Trajectory &)
4: Cross (*this)
5: SelectClusters(SpacePoint &)

Fig. 4. Select clusters compatible with a trajectory.

Main detector components (`MainComponent`) collaborate with active detector units (`DetectorUnit`) to select clusters compatible with the current candidate track represented by geometric parameters (`Trajectory`). The double dispatch idiom is used to intersect all possible `DetectorUnit`'s shape properties, the `Surfaces`, and the trajectories built during the pattern recognition: whatever the particular surface crossed by the particular trajectory, the proper crossing algorithm is applied (messages 3 and 4). The pre-selection of clusters to be analysed is performed around the impact point (`SpacePoint`) on the detector surface. This collaboration is established at the level of abstract classes so as to extend the capabilities without interfering with the existing design and code. New geometrical shapes and trajectories will find the collaboration ready for them to cooperate. Double dispatch idiom is sometimes called also "double polymorphism" or "multi-method".

A slightly less simple example is the collaboration to determine the new set of possible detector units that the candidate track may cross with a subsequent recognition step and the subset of each detector unit clusters to be analysed.

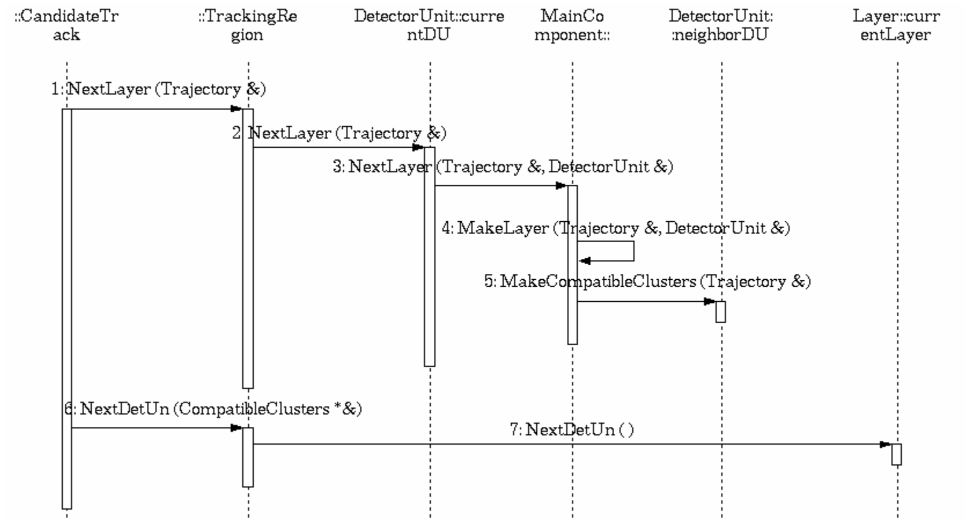
::CandidateTr ack
::TrackingRe gion
DetectorUnit::curre ntDU
MainCo mponent::
DetectorUnit: :neighborDU
Layer::curr entLayer
1: NextLayer (Trajectory &)
2: NextLayer (Trajectory &)
3: NextLayer (Trajectory &, DetectorUnit &)
4: MakeLayer (Trajectory &, DetectorUnit &)
5: MakeCompatibleClusters (Trajectory &)
6: NextDetUn (CompatibleClusters *&)
7: NextDetUn ()

Fig. 5. Build logical layer of next reachable detectors.

Here the `CandidateTrack` object is ready to proceed with a new step in the reconstruction process. To do this it needs to know from the detector layout which are the detector units it may cross with the new step, and which are those clusters that may be along its trajectory. The use case shows two sequences of collaboration. The first one builds a `Layer` object, the logical collection of detector units and clusters that are reachable by the candidate track. This collaboration is triggered by the candidate track by sending the message `NextLayer(Trajectory &)` to the `TrackingRegion` object. This latter has knowledge of all the logical layers built in the previous steps. Once the control returns to the candidate track, the second collaboration takes place: the tracking region changed state so that it may now provide the next detector unit and clusters to process (message 6).

In the previous example the candidate track also controls the sequence of events since the next detector unit (message 6) is actually provided by the newly formed layer (message 7). However, the candidate track has no possible way to know how the `TrackingRegion` deals with `Layers` and detector objects, nor has it any knowledge of this taking place.

For the purpose of system evolution, what happens beyond the `TrackingRegion` object may undergo drastic changes without the candidate track, or its collaboration with the tracking region, even noticing that. If the evolution of the system should require that, the ability of the detector objects to determine the subsequently reachable ones, given a geometrical trajectory, may be reused in a different context. Delegation permits the encapsulation of the object collaborations for possible reuse.

5. Development Process

System development is indeed a complex task and in order to achieve the sought result, complexity must be handled in an organised way. By working with different models of the system to develop, complexity is introduced gradually. It is important to define a process since it instills a discipline into the development of software systems, defines the products that serve as communication among the members of a development team, and defines the milestones needed by management to measure progress and to manage risk.

We see the development process as based on architecture, method, and the process. The following observations hold true:

- The process must yield the same deliverables, irrespective of which individual performs the job
- The volume of output does not affect the process
- It must be possible to allocate parts of the process to several independent teams
- It must be possible to make use of predefined building blocks and components

The development process is seen as consisting of five distinct seamless models: **requirements** or **conceptualisation**, **analysis**, **design**, **implementation**, and **testing**. By seamless we mean that one has to be able to get from concepts and objects in one model to concepts and objects in another model. This is crucial for a successful development process since the result must be repeatable. **Maintainance** of the system is necessary once the system is, for so to say, in production. Maintainance task is simplified if we have traceability between the models.

5.1. *Requirements or Conceptualisation*

In requirements we specify *what* the system has to offer its users. The idea is to capture the requirements of the system from user's perspective. This activity is often conducted in close relation with the end users and addresses questions to unambiguously describe how they will use the system. The consensus once achieved, the system is structured from a logical perspective that aims to a robust and adaptable form. In our experience the use cases have played an important role in this conceptualisation activity.

5.2. *Analysis*

The analysis must be carried out in an ideal world and independent of the implementation environment. Working in the ideal circumstances reduces complexity and allows to focus the effort on building a logical structure that is stable, robust, and flexible. The model resulting from the analysis should not be overly elaborate to permit adaption due to design and implementation choices.

Changes are unavoidable, and even welcomed: too formal an analysis model results into an inflexible architecture that will not cope with the evolving requirements of the problem and user domains. Especially the implementation environment will change during the software life cycle and it is undesirable that current circumstances affect the system structure. One should work on analysis long enough to understand the system completely, but not so long as to consider details which will be modified during design.

5.3. *Design*

Design defines *how* to realise the analysis model. This activity formalises the analysis and specifies the building blocks of the system. These will realise the functional points required to fit the system requirements. Subsequently, the building blocks will be implemented. In this model we address questions, for example, on how to integrate a OODBMS or how to handle a distributed environment. Where the analysis model cannot be directly implemented, the design should be.

5.4. *Implementation*

When the above decisions are made and the system further formalised, the implementation model is developed: this is the actual code to be produced. Hence, we may say that in design and implementation the ideal world of analysis will be replaced by the additional requirements arising from the development environment.

5.5. *Testing*

The system is checked to make sure that the original path traced in the models is not lost and that the performances meet the requirements. This usually involves documentation of the test specifications and the test results. Many of the foreseen tests, especially at code level, may be performed making use of dedicated tools. In the end, the system should be validated to determine whether it performs accordingly to the user requirements and whether its documentation describes it from the user's perspective.

Jacobson also points out to use cases in the testing process: "... For the first time several classes, block, service packages and subsystems are brought together and therefore the testing should concentrate on this. Each use case is initially tested separately. The use cases constitute an excellent tool for integration test since they explicitly interconnect several classes and blocks. When all use cases have been tested (at various levels) the system is tested in its entirety. The several use cases are executed in parallel and the system is subject to different loads."

5.6. *Maintenance*

Maintenance manages the post-delivery evolution. Normally, the issues that require changes in the architecture of the system are not addressed; rather localised changes will be made as new requirements and/or defects are found in using the system in its final environment.

6. System life-cycle

The system is gradually refined in cycles using the models mentioned above. The models form a sound base on which the complexity of the system is managed as it is introduced step by step by focusing on the more important aspects. These models should not be viewed as *sacred* or untouchable: they are not the final answer! An evolutionary approach with gradual refinement of the system would not be compatible with such an attitude.

System development implies also the progressive changes made as new and modified requirements are imposed to the product. In a rapidly changing environment the development process should support an evolutionary approach to handle such changes during the construction of a specific version of the product. The evolutionary approach also features beneficial early reactions to feedback the subsequent development cycles. To complement the process, early prototyping can be used to explore and prove uncertain features well ahead in the life cycle of the system.

Each developer will deal with these activities together with the members of the same team. A software project often has several teams, and certainly this is the case with the software to be produced for LHC experiments. This picture calls for managerial issues that have been recognised and studied since early ([Brooks, 1972]).

The life-cycle notion depicts the activities and measures project progress from the requirements specification to the development of the software and submission of the finished product. This notion has been formalised in different development structures or models[a], where all their phases are somewhat linked together in sequence. A first example of traditional life-cycle is the waterfall model that established the basis of the formalism.

6.1. *The Waterfall model*

In the waterfall model (Fig.6) the development is split into sharply defined phases each constituting the information source to the subsequent one. Variations aimed at improving this over simplified life-cycle exist such as the back-stepping, as shown in the figure, or the V model where emphasis is placed on the preparation of the integration and validation phases. The problem with the waterfall model is its crisp separation between modelling theory and the "real life" environment. It is

[a]The term *model* is used with a wide meaning.

in general too late when exceptions and unmatched assumptions are found: the implementation diverges from the original model, often in an uncontrolled way.

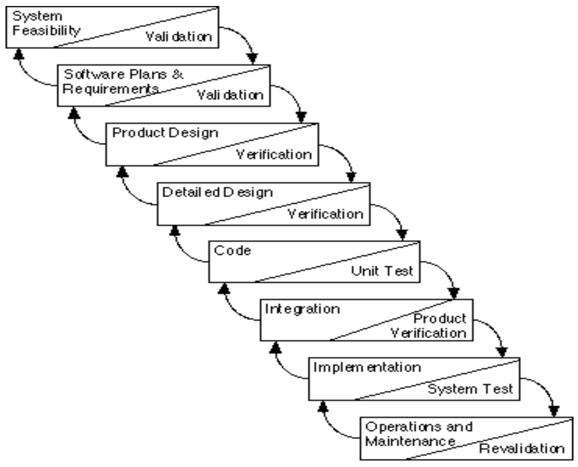

Fig. 6. The waterfall model.

The criticisms to the waterfall model may be resumed to ([Humphrey, 1989]):

- It does not adequately address changes
- It assumes a relatively uniform and orderly sequence of development steps
- It does not provide for methods such as rapid prototyping

Waterfall-like models also rely on the assumption that the system is fully specified from the beginning and that the development team will not be faced with obscure areas or possibly different understandings of the problem because of new insights, during its entire life-cycle. The fundamental principles on which waterfall-like models are based are nevertheless valid.

6.2. *The Spiral model*

Improved strategies have been adopted where the project is broken down into subparts to which an entire life-cycle is applied. Clearly identified parts are developed first and the insights achieved are used for less evident subparts. The system is built incrementally part by part. This approach has been introduced by Boehm ([Boehm, 1985]) as the Spiral Model life-cycle (Fig.7). Boehm introduced testing, prototyping and risk analysis for the obscure areas of a software project.

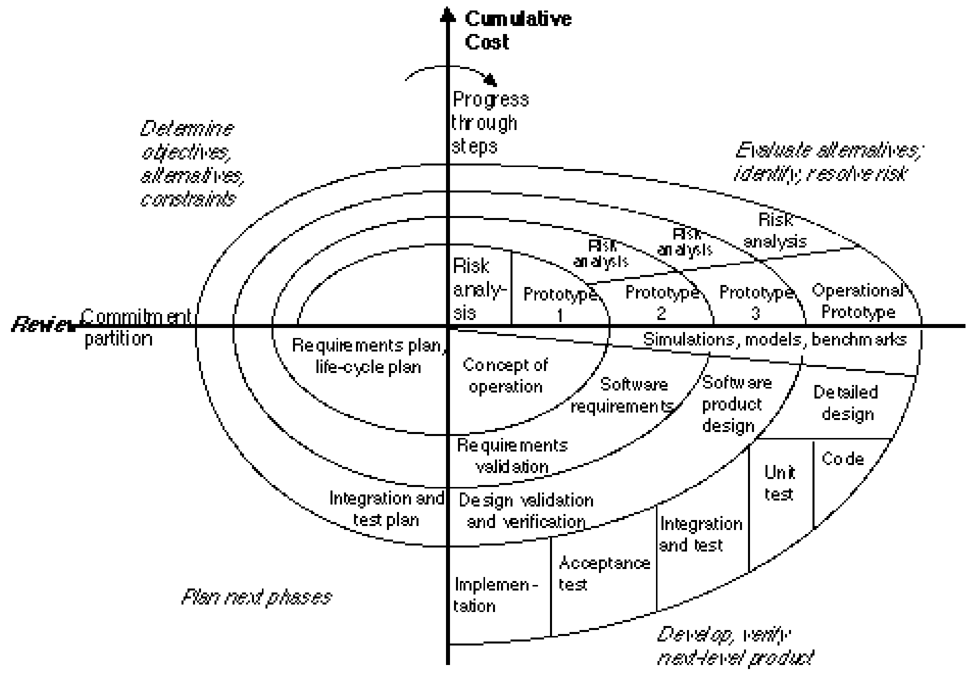

Fig. 7. The Spiral model.

The main conclusion from previous studies and works is that fundamental phases exist[a] and that a strategy must be adopted to effectively deal with them.

Traditional life-cycle models are said to be "requirements-driven" and may be resumed with the following process ([Booch, 1996]):

(i) Enumerate all of the system functions
(ii) Design components for each of these threads
(iii) Implement each component
(iv) Integrate

These models suite well with SA/SD developments techniques. On the contrary, modern OO methods emphasise the incremental, iterative, evolutionary, concurrent and situational nature of software development.

[a]Conceptualisation, analysis, design, and so on.

6.3. *The Evolutionary OO model*

System development involves also progressive changes as new and modified requirements are imposed on the product. As said before, in a rapidly changing environment, as HEP is, the process should handle the changes during the development of a specific version of the application. Traditional models are easily thrown out of balance when the problem domain evolves and puts changes upon the requirements.

Evolutionary development strives for a well-designed software architecture, in terms of categories of classes applying the Open-Closed Principle and featuring patterns of collaboration. A well-designed architecture shows itself adaptable to changes arising from varying conditions and requirements, either new or modified.

In evolutionary development the system is built and delivered as a series of partial, but increasingly complete, implementations. Software is integrated early and often, at each evolution instead of at the end of the project. The nature of the evolutionary process of OO development means that rarely, if ever, a single "big bang"; integration event occurs. Each release evolves from an earlier stable release. The system deliberately satisfies fewer requirements at the beginning but is constructed to facilitate the addition of new requirements, thus achieving higher adaptability. Frequent integration reduces risk by exposing it early in the project lifetime. This strategy accelerates the discovery of architectural and performance problems in the development process.

Software systems have technical as well as non-technical risks. Technical risks in OO systems include problems such as the selection of an architecture that features the best in terms of usability and flexibility. Another example is the choice of "mechanisms"; that yield acceptable performance while simplifying the system's architecture itself. Non-technical risks concern supervising the delivery of software, from a third-party or in-house, and managing the relationship between the final user[a] and the development team to discover the system's real requirements during analysis. Because we can get a working system from the beginning of the life-cycle, we may better keep it on-track with the requirements and the needs well before the project completion. These often constitute a living document.

> *An architecture driven development has all the benefits of a requirements-driven style, as well as the favourable characteristic of encouraging the creation of resilient frameworks that can withstand shifting requirements and technological calamity.* [Booch, 1996]

It is important to view object-oriented development not merely in terms of a relatively informal coding practice, but rather at least partial as a life-cycle process.

7. We Need a Software Process

Establishing a well defined software development process is at the heart of future CMS software effort. Current experience shows the importance of:

[a]The physicist in our case.

- Iterative development (to reduce risk by exposing it early)
- The software architecture (to achieve flexibility)
- Software reuse (through focusing on interfaces)
- The use of object-oriented methods in day-to-day analysis, design, implementation, and maintenance

In an evolutionary architecture-driven development process we specify the objectives of a software system through a collection of scenarios where responsibilities are shared among the participating objects. An architecture is built and validated to support the scenarios while exploiting the common collaboration patterns found there. The system is built and the architecture evolved making mid-course corrections to adapt to new requirements as they are uncovered.

The most striking feature of the structure of an architecture-driven project is that its components tend to map to the abstractions we find in the real world, hence filling the knowledge gap between the problem domain and its representation as software.

7.1. *The Booch Process*

The Booch approach to an evolutionary development process is at two different levels: a macro-process, that addresses organisational and managerial issues at each evolution, or release of the system, and a micro-process that covers the everyday activities of the developers.

7.1.1. *The macro-process*

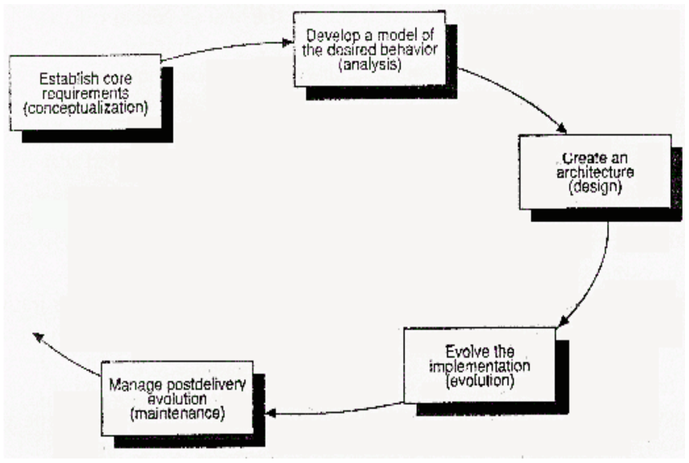
Establish core requirements (conceptualization)
Develop a model of the desired behavior (analysis)
Create an architecture (design)
Evolve the implementation (evolution)
Manage postdelivery evolution (maintenance)

Fig. 8. The macro-process, from [Booch, 1994].

The macro-process is important to identify problems early in the life-cycle and to respond meaningfully to the risks before they jeopardise the success of the project.

Project planning involves scheduling the deliverables in the macro-process. Between evolutionary releases, the management must assess the imminent risks to the project, address the resources so as to attack those risks, and then manage the next iterations of the micro-process that will yield a stable system.

7.1.2. *The micro-process*

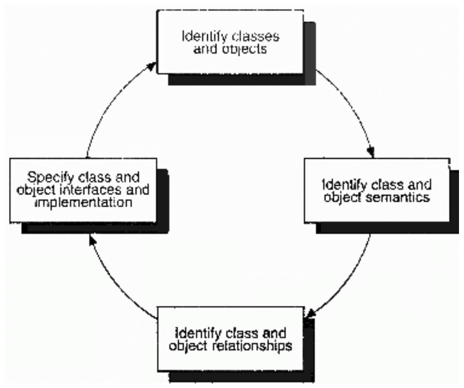
Identify classes and objects
Identify class and object semantics
Identify class and object relationships
Specify class and object interfaces and implementation

Fig. 9. The micro-process, from [Booch, 1994].

The micro-process of object-oriented development is inherently subjective and requires the macro-process as a driving force. The process is designed to lead to completion by providing a number of tangible products that management can study to judge the health of the project, as well as controls that allow management to redirect resources as necessary.

The Booch approach has been followed in the development of the CMS Pilot Project ([Innocente and Marino, 1995] and [Bos et al., 1995]).

8. Project Management

When developing software systems we must also consider sound managerial practices with regard to subjects like staffing, release management, and training. To most physicists, these are non-issues since the software will be there anyway, somehow, almost by magic. Unfortunately it is almost never the case: these are realities that must be faced to build successful complex software systems. With an iterative and evolutionary life-cycle it is evident that the project's activities have to be managed.

8.1. *Task Planning*

The basic practices of software development management, e.g., task planning, walk-throughs for analysis and design validations, code inspections, are unaffected by object-oriented technology. These activities in general require the development team to meet and communicate improvements. Both formal, scheduled meetings and informal brainstormings are necessary.

Some minimal frequency of meetings is needed for communication within the team; too many meetings, on the contrary, would destroy productivity. In the CMS OO activities, it has been reasonable to have weekly team meetings to discuss completed work and incoming mailstones. Unnumbered chats, exchanged ideas and joint mumbling proved invaluable.

Object-oriented software development requires that individuals have unscheduled time to think, invent, develop, and meet informally other team members to discuss detailed technical issues. The management team must take into consideration this unstructured time.

Meetings provide an effective vehicle for tuning schedules in the micro-process and for gaining insight into potentially complex areas. These meetings can result in small adjustments to the tasks to ensure the progress of the ongoing work. In our projects we will find that developers cannot wait for other team members to stabilise their parts of the architecture. In object-oriented systems, classes and mechanisms heavily affect the system architecture: development can stall if certain key classes are still in the clouds (no reference to Booch notation).

Task planning involves scheduling the deliverables in the macro-process. Between evolutionary releases, the management team must assess the imminent risks to the project, allocate the resources to attack those risks, and then plan the next iterations of the micro-process that will yield a stable system. Task planning at this level most often fails because of too optimistic schedules. Booch suggests that in order to develop schedules in which the team can have confidence, the management must devise multiplicative factors for the developers' estimates. Anyway, management must realise that effective planning is a skill that is acquired only through experience.

OO helps in this respect because an iterative and evolutionary life-cycle requires many intermediate milestones to be established early in the project. These milestones can be used to meet schedules and priorities. As evolutionary development proceeds, management will gain a better understanding of the real productivity of each of its developers over time, and individual developers can gain experience in estimating their own work more accurately.

Incidentally the same kind of lessons apply to tools: with early delivery of architectural releases, OO encourages the use of tools early and leads to the identification of their limitations before it is too late to change them.

8.2. *Walkthroughs - A validation procedure*

Walkthroughs are another established habit to employ. Management should balance between too many and too few walkthroughs, keeping in mind that it is simply not practical to review every line of code. In our projects we should regularly conduct formal reviews on scenarios as well, and on the system's architecture, with many informal reviews focused on smaller tactical issues.

Scenarios are a primary product of the analysis phase of OO development and serve to capture the desired behaviour of the system in terms of its functionalities.

Formal reviews of scenarios are led by the team's analysts together with the domain experts or other end users and are witnessed by other developers. These reviews are best conducted throughout the analysis phase, rather than in one massive review at the end of analysis, when it is already too late to do anything useful to redirect the efforts.

Our experience shows that even non-programmers can understand scenarios presented through scripts or through the formalisms of interaction diagrams. Ultimately, the reviews help to establish a common vocabulary among the developers and the users.

Architectural reviews should focus on the overall structure of the system, including its class structure and mechanisms. As with scenario reviews, architectural reviews should be conducted throughout the project and led by the project's architect or other designers. Early reviews focus on architectural issues that have to stabilise, whereas later reviews focus on particular class categories or specific mechanisms, also called object interactions.

The main purpose of these reviews is to validate designs early in the development. A secondary purpose is to increase the visibility of the architecture among the team in order to create opportunities for discovering patterns of classes or collaborations of objects, which then can be exploited to simplify the architecture itself.

Informal reviews may be carried out weekly and generally involve peer-to-peer review of particular clusters of classes or lower-level mechanisms. The main purpose of these reviews is to validate tactical decisions.

8.3. *Release Management*

From the perspective of the users of the system, the macro-process in the OO evolutionary development generates a stream of executables, each with increasing functionality, and eventually evolves into the final system. From the developers' point of view usually many more releases and prototypes are constructed.

In larger projects, internal releases of the system could be produced every few weeks. A running version along with its associated documentation can be shipped to the users for review every few months, according to the needs of the project.

8.4. *Configuration Management and Version Control*

Consider the problem from the perspective of an individual developer who is responsible for the implementation of a particular category. He usually has a working version of that subsystem and a version under development. To proceed with the development, at least the interfaces of all imported subsystems must be available. As this working version becomes stable, it is planned for integration.

Who will be responsible for collecting the compatible subsystems for the entire system? Probably when the projects will get considerable size a dedicated team will be devoted to this task. Eventually, this collection of subsystems is frozen, put on the base line, and made part of an internal release. This internal release becomes the current operational one visible to all the developers who need to further refine their particular parts. In the meantime, the individual developer can work on a newer version of his subsystem. In this way, development can proceed in parallel, with stability possible because of the well defined and the well guarded subsystem interfaces.

At any point in the evolution multiple versions of a particular subsystem can exist: one version for the current release under development, one for the current internal release, and one for the latest customer release. This situation explicitly needs tools for configuration management and version control.

Implicit in this model is the idea that a category or a cluster of classes, not the individual class, is the unit of version control. Our experience suggests that managing versions of individual classes is too fine a granularity because no class tends to stand alone. The CMS Pilot project has been put under versions control using the categories as controlled unit.

The concepts of configuration management apply not only to the source code, but also to all the other deliverables of the OO development, such as requirements, class diagrams, object diagrams, documentation files, and so forth.

8.5. *Technology Transfer*

Learning object-oriented programming can be more difficult than just learning another programming language, often because a different perspective is involved rather than a different syntax in the same framework. Indeed, you must learn a new way of thinking about programming.

We have to develop this object-oriented mind-set by providing formal training to both developers and managers in the elements of the object model. An important step we made is to use OO first in a low-risk project and allow initial developers to make mistakes. In the future, use these team members to seed other projects and to act as mentors for the object-oriented approach.

9. Deliverables

The development of a software system requires much more than writing plain source code. Certain deliverables of the development process provide the means to give the management team and the users information about the progress of the project. Documentation of the analysis and design decisions must be produced also to benefit the eventual maintainers of the system. The products of object-oriented development essentially are sets of:

- Class diagrams
- Object interaction diagrams

Object interaction diagrams denote scenarios[a] conceived in order to fit the re-

[a]Patterns of object collaborations.

quirements, while class diagrams represent key abstractions that form the vocabulary of the problem domain and support the implementation of the mentioned scenarios. As a whole these diagrams offer the possibility to trace back to the system requirements.

Module and process diagrams are additional parts of the Booch method. Process diagrams show how processes are allocated to processor in the physical model of the system, processors and devices that serve as the platform for the execution of the system concerned; module diagrams show the allocation of classes and objects to modules of the system such as subsystems, specifications, source bodies, and the main program.

Not all these modules are supported by any language, e.g., C++ only supports the concept of files, specifications (the class declarations), source bodies (definitions), main program, and their dependencies. Each of these modules represents the implementation of some combination of classes and objects, which are in turn found in class diagrams and object diagrams.

The documentation of a system's architecture and implementation is important, but the production of documents should not really drive the development process. Documentation is an essential part of the system development although it has to be seen in the right perspective in the process; it is a support product. It is also important to remember that documents are "living" things, hence they should be allowed to evolve together with the iterative and incremental evolution of the project.

Together with the generated code, delivered documents serve as the basis of most of the formal and informal reviews (walkthroughs, code inspections and the like).

9.1. *What must be documented?*

End-user documentation must be produced to instruct the user on the operation and installation of each release of the system. In addition to that, analysis documentation must be produced in order to capture and store the semantics of the system's function points as viewed through the scenarios.

Architectural and implementation documentation must be generated to communicate the overall vision and the details of the architecture to the architects and the developers and to store information about all relevant strategic decisions - the fundamental *whys* of decisions - so that the system can be adapted and evolved in the case the reasons should change.

In general, the essential documentation include the following:

- The high-level system architecture
- The key abstractions and mechanisms in the architecture
- The scenarios that illustrate the built-in behaviour of key aspects of the system.

The addition of the word *key* means that I do not expect every single detail of the system to be documented, especially the irrelevant internal details of the implementation.

The worst possible documentation we could produce to describe an object-oriented system is if we limit ourself to only the description of the methods on a class-by-class basis, and to the purpose of each single class. This approach would generate a big amount of quite useless documentation that no one really reads. On the other hand it would present an object-oriented system as a collection of island-classes. This misses completely the goal of documenting the important architectural issues that go beyond the individual classes, that is, the collaborations between classes and objects.[a]

It is far more effective to document these higher-level structures expressed in diagrams of the notation that have no direct counterpart in the programming language adopted to implement the system.

10. Final remarks

Software development is not an easy task and object-oriented technology may constitute a real help. As we see things today, the problems we face now will appear trivial in the next ten years. Traditional approaches are already facing their limits in handling the present complexity. Will OO be able to be a clean solution to that? We can't honestly say, but we cannot deny that OO will be an important step on that way, a step we cannot afford if we stick to the older techniques.

The discipline of Object-Oriented analysis and design is more complex than other software disciplines. The complexity arises from variety: we may invent millions of different perfectly valid objects, and the classes from which they are created. Also, great variation of relationships exist between them and the way we may make them to collaborate with each other. We accept to face the complexity of the OO discipline to achieve a simpler, more flexible and robust design.

To add to this picture, each different object, class, relationship has its own proper semantics. This complexity makes the discipline of OO analysis and design harder to grasp , learn, and master than any other. But, and from this come the benefits, OO has an unprecedented semantic richness and expressiveness. Software maintains a closer mapping with the real world, reducing the gap between the understanding of the problem domain and its software counterpart. The problem and the software solution may use now the same dictionary and expressions.

[a]*No object is an island*, G. Booch.

Acknowledgements[a]

We wish to thank our colleagues of CMS and RD41 for their contribution in terms of discussions, suggestions, and the explanation of their different perspectives. We are indebted to V. Innocente, W. Jank, and M. Pimia - without their support and encouragement this work would have not been written. We like to mention F. Rademakers and S. Ravndal with whom the discussions and querelles have continued on the gOOlf courses. Finally special thanks to L. Tuura for his help in fine tuning the text and permitting the readers to get the intended messages.

References

[ACM, 1987] Association for Computing Machinery, OOPSLA'87 Conference Proceedings, special issue of SIGPLAN Notices, Vol. 22, No. 12, December 1987.

[Boehm, 1985] B. W. Boehm, *A Spiral Model of Software Development and Enhancement*, Proceedings of International Workshop on Software Process and Software Environments, Coto de Caza, Trabuco Canyon, California, March 27-29, 1985.

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design - With Applications*, Second Edition, Benjamin/Cummings, Menlo Park, California, 1994.

[Booch, 1996] G. Booch, *Object Solutions - Managing the Object-Orient ed Project*, Addison-Wesley, Menlo Park, California, 1996.

[Booch and Rumbaugh, 1995] G. Booch and J. Rumbaugh, *Unified Method: User Guide, Version 0.8*, Rational Software Corporation, Santa Clara, California, 1995.

[Bos et al., 1995] Moose Project, *Status Report of Moose - an Object Oriented Approach to Software Development for LHC Experiments*, CERN/LHCC 95-60.

[Brooks, 1972] F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley, Menlo Park, California, 1972.

[Coleman et al., 1994] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Cook and Daniels, 1994] S. Cook and J. Daniels, *Designing Object Systems - Object Oriented Modelling With Syntropy*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[a]For interaction diagrams we used ©Rational Rose/C++ v. 3.0

[Gamma et al., 1994] E. Gamma, R. Help, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachussetts, 1994.

[Humphrey, 1989] W. Humphrey, *Managing the Software Process*, Addison-Wesley, Menlo Park, California, 1989.

[Innocente and Marino, 1995] V. Innocente and M. Marino, *An Object-Oriented Approach to CMS Reconstruction Software*, CHEP,95 Conference Proceedings, Rio de Janeiro, Brasil, September 1995, CMS TN/95-139 SW.

[Jacobson, 1987] I. Jacobson, *Object-Oriented Development In an Industrial Environment*, OOPSLA'87 Conference Proceedings, special issue of SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 183-191.

[Jacobson et al., 1992] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts,1992.

[Jacobson et al., 1995] I. Jacobson, M. Ericsson, and A. Jacobson, *The Object Advantage: Business Process Reengineering With Object Technology*, Addison-Wesley, Reading, Massachusetts, 1995.

[Liskov, 1988] B. Liskov, *Data Abstraction and Hierarchy*, SIGPLAN Notices, Vol.23, No.5, May, 1988.

[Martin, 1995] R. C. Martin, *The Liskov Substitution Principle*, C++ Report, March 1996.

[Meyer, 1988] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Meyers, 1992] S. Meyers, *Effective C++ - 50 Specific Ways to Improve your Programs and Designs*, Addison-Wesley, Reading, Massachusetts, 1992.