Using object oriented technology to measure a software process*

Amund Aarsten, Maurizio Morisio

Dipartimento di Automatica e Informatica

Politecnico di Torino

Corso Duca degli Abruzzi, 24, 10129 Torino, Italy

e-mail: {amund, morisio}@polito.it

fax: + 39 11 5647099

voice: + 39 11 5647033

Abstract

This paper reports on how object oriented concepts and technology have been applied to measure a software process in the ESSI AEFTA (Application Enabler Technology for Factory Automation) application experiment.

Measurement is considered as a full scale project consisting of 1) modelling the software process, 2) defining goals for the project, 3) formalizing goals in process measures defined on the process model, 4) designing the technological support for measures (procedures and tools, repository), 5) implementing the technological support, 6) implementing the measurement process and 7) assessing the measurement process, modifying it, if needed.

The object oriented (oo) approach has been applied to 1), 4) and 5). The process model is defined using OMT; process measures are defined using a query language to express queries on OMT object models; the repository for measures is obtained by automatic translation of the OMT process model into a set of persistent C++ classes; process measures are manually translated in C++ member functions of the persistent classes.

The advantages of applying the oo approach are discussed: sharing of tools, training, know-how and mindset with the rest of the project; ease in changing measures to adapt to changes in the process, thus the possibility of applying an evolutionary incremental lifecycle to the measurement process; lower effort in the implementation of the technological support for measurement.

Keywords

Object oriented approach, software measurement, software metrics, process modelling.

*This work was partially supported by the European Commission, DGIII/A, under contract ESSI AEFTA n. 10070.

1 INTRODUCTION

It is generally recognized that the best way to assess and improve a software development process is to measure it, that is to measure effort, cost, duration, faults, failures and changes (Fenton 1991, Hetzel 1993). When collected over a number of projects in the same context, these measures can be used to build prediction models for future projects (Jones 1991). Also the product must be characterized, internally (size, complexity, modularity) and externally (functionality, usability, efficiency, reliability, portability, maintainability).

The actual situation in software industry reveals a limited use of measurement. Measurement methods (such as GQM (Basili and Rombach 1988), AMI (ami 1992) and others reviewed by Roche (1994)) have been proposed to structure bare measures in a framework and propose a process to simplify their application.

Our approach to the measurement problem is that measurement should be based on a process model, and that the process model, the data repository and the whole measurement process should use object oriented concepts and technology.

1.1 The process model as a base for measurement

The issue of process modelling is receiving growing attention in the software community. The process model is seen as the starting point to analyse, improve and enact the process, but the need of strict coupling between process modelling and process measurement has not yet clearly emerged.

Basili and Weiss (1984) consider the measurement process and its validation, but do not couple the measurement process and the surrounding software process. Bache and Bazzana (1994), Hetzel (1993) cite a process model but do not define nor use it. Pfleeger and McGowan (1990) associate sets of measures with the levels of the Capability Maturity Model but do not define measures on a process model. Wolf and Rosenberg (1993) define (implicitly) a process model in terms of production of events, then collect and analyse them to assess the process. Cook and Wolf (1994) continue in the same line by defining measures of distance of sequences of real events from process model events.

Matsumoto et al. (1993) are the first authors, to our knowledge, to use a process model to define and collect process measures. The model is represented by Petri nets; duration measures are coupled with transition firing. The Petri net representation allows enacting the process and automatic collection of the measures. The drawback of this approach is in the use of Petri nets, which are not adequate to express structure and static relationships, nor to model complex systems such as a real world process model.

Our thesis is that the collection of valid measures relies on a clear, unambiguous, easy to understand and communicate definition of the process (a process model, according to the terminology of the software process community (Feiler and Humphrey 1993)). Further, the definition of measures should be integrated in the process model. Let us examine why.

Usually when measures are collected in current software engineering practice, three situations can occur.

1. The process is not defined, measures are applied and their definition is taken from the literature. A definition of a measure relies on the identification or definition of the attribute of an entity, or the entity to be measured (see Fenton's framework for

measurement in (Fenton 1991)). An entity defined in literature and the equivalent entity in the context of a process can differ, resulting in mismatches between the definition of measures and their application in the actual process context. For instance, to compute the design effort how is design defined, does it consists of the initial design only, or of rework to fix faults too? And how is a fault defined? Is an incoherence between design documents a fault or not? If the design documents are not yet validated, is it still a fault? And when is a document considered to be validated? *Standard* entities, such as design and fault, in fact rely heavily on their context. Since process measures are in many cases collected manually by different people, it is plausible that each person give a different interpretation to entities, producing unreliable measures.

2. The process is defined in natural language, process measures are applied and their definition is taken from the literature. The risk of mismatches still exists, but it is lower, since the definition of the process prescribes coherency checks. The real problem here is the complexity of the process, which is not adequately modeled in natural language, with the well known drawbacks in analysis and communication. This leads again to the risk of different interpretations by different people, producing unreliable measures.
3. The process is defined in a semi-formal or formal language. Entities are defined in a language suitable to represent them and to tackle complexity. The risk of different interpretations is greatly reduced. Now the problem lies in the separation between process definition and measures definition which can lead to incoherencies. Adding a measures-process validation task is one solution, better is to merge measures and process definition, that is to define measures as an extension of the process model.

1.2 The object oriented approach and measurement

Given the need to define measures on a process model, which is the best formalism for it ?

If "software processes are software too" (Osterweil 1987) and if measurement is part of the software process, then measurement is software, too. So why not to apply to measurement the object oriented (oo) approach, to exploit its well known advantages in this area too ?

The majority of works in the process modelling domain and all commercially available process support tools do not take this point of view and use activity or dynamics oriented formalisms such as Petri nets (see for instance Curtis (1992)). The rationale of this choice is that the model is used to enact the process. From a measurement point of view, entities and the static relationships among them are more important, thus favouring an oo view.

Our choice for the process model language was OMT (Rumbaugh et al. 1991). We preferred the general OMT over specialized process modelling languages because: it allows the description of the information part of the process along with the dynamic aspect; it allows to structure easily the model of a complex system. Last but not least, it is used in the application experiment to develop the product, therefore allowing to share tools and training.

The definition of the process model and of measures on it corresponds, in a software project, to the specification of requirements phase. Continuing in this correspondence, the design and implementation phase consist of designing and implementing tools and procedures to collect, store and analyse measures. If the object oriented approach is used, these tasks are partially automated. For instance in the AEFTA (Application Enabler Technology for Factory

Automation) application experiment, the tool used to define the OMT model supports the semi-automatic definition of the measurement database and forms on the database.

2 THE PROJECT

The AEFTA application experiment, which involves Digital Equipment Corp., Politecnico di Torino and Syco, develops a factory automation product using BASEstar (DEC 1990) and object technology. BASEstar is, in factory automation terminology, an application enabler that provides a number of basic services such as support for communication with field devices, support for building GUIs, libraries of emulators of programmable logic controls (PLC) and other field devices. Object technology means C++ and G++ (Menga 1993), which is not the gnu compiler but a homonym CASE tool supporting oo analysis and design via a modified version of OMT, a concurrency/distribution model, and generation of code.

The product now developed in object technology already exists in functional technology (C language and BASEstar). The experiment must answer to the questions: is the oo product of the same quality as the functional product, has the oo product the same functionality, how much reuse is made in the oo product, how much this reuse did cost.

To answer these questions a measurement process is defined and implemented, which uses oo concepts and technology. The measurement process is described in the following section.

AEFTA is an application experiment in the ESSI (European Software and Systems Initiative) initiative, sponsored by the European Community to advance the state of the art in software production.

3 THE MEASUREMENT PROCESS

The measurement process consists of 1) modelling the software process using OMT, 2) defining goals for the project, 3) formalizing goals in measures defined on the process model, 4) designing the technological support for measures (procedures and tools, repository), 5) implementing the technological support, 6) operating the measurement process, 7) assessing the measurement process, modifying it, if needed. In the following the first five steps, and in particular steps 4 and 5, will be described. Step 1) in section 3.1, 2) and 3) in 3.2, 4) in 3.3, 5) in 3.4. For a complete description of the measurement process see (Morisio 1995).

3.1 Modelling the software process

The first activity is the definition of the process model, since measures are defined on it. The model should be a snapshot of what actually happens, and not a model of what should be. According to Feiler and Humphrey (1993) a descriptive and not a prescriptive process model. Therefore no emphasis is given to the lifecycle used (waterfall, spiral, or else).

The process is modeled in OMT, using mainly the object model, in which activities, products and roles are modeled as classes. The dynamic model is limited to some classes having rich or complex behaviour. The rationales for these choices are explained in section 4.

Now we present and comment a reduced version of the process model (refer to (Aefta 1994) for the complete version). **Boldface** denotes classes and associations of the model.

Figure 1 Process model, overview.

Figure 2 Process model, specialisation of **Process Item**.

Figure 3 Process model, specialisation of Process Activity.

Process Item

The basic classes of the model are **Process item** and **Process activity** (see Figure 1). A **Process item** models whatever item is produced during development and specializes in **Product component** and **Process document**. A **Product component** relates to the product and can be (see Figure 2) **Analysis document**, **Logic Design document**, **A_class**, **LD_class**, **PD_class**, **C++ class**. A **Process document** can be (see Figure 2) **Process model**, **Measurement plan**, **Measurement report** and so on.

Process Activity

A **Process activity** (see Figure 3) models the time spent by one person doing something in the project under consideration. It is characterized by its duration and by the fact that the person works alone (**type = individual**) or with other people (**type = group**). Group activities are characterized by the communication means used (**means** attribute). Activities can be **creation/modification** (creative work involving modification), **verification/validation** (inspection work not involving modification), **research/browsing** (search work not involving modification nor inspection), **study** (self study or training), **communication** (teaching or oral presentation), **travelling** (displacement outside the normal working site to perform an activity in the context of the project).

Activities are **neutral**, since they do not specify the object on which work is done. The associations **Process activity works on Process item** and **Process activity works on External product model** this information.

Since associations are inherited too, **works on** holds between any specialization of **Process activity** and any specialization of **Process item** or **External product**. For instance some work on the analysis document is modelled by an instance of **creation/modification**, an instance of **Analysis document** and a link between them, instance of **Process activity works on Process item**.

The strong link between activity and what is acted upon is contained in the cardinality one and only one between **Process Activity** and **Process item**, that means no activity can exist without a connected process item.

Process Phase

The traditional definition of phases, such as analysis, design, implementation, testing and the like, are represented as **Process Phases**, with attributes **initiated** and **terminated**. They model the milestones of the project.

A **Process Phase** specializes in **Analysis**, **Logical design**, **Physical design**, etc.. The definition of each specialization is delegated to specializations of association **Process Phase works mainly on Process Item** (see Figure 1). For example **Analysis works mainly on Analysis Document** defines **Analysis**.

Process Phase is an aggregation of **Process activity** models that in a phase many concurrent activities happen, while activities have no concurrency inside. Since each activity **works on** an item, in a phase work is made on many items. The document that is the focus of the phase is defined by **works mainly on**.

The description of a phase via the document on which it works follows an oo style; (objects, such as the structure and content of documents, are privileged against functionality, such as what is made during phases) and has advantages in the definition of measures (see 3.2 for definition of measures and 4 for a discussion).

External Product

An **External product** is an object which is not developed in the software process, but used in it (association **works on** in Figure 1). Among other things, the **External products** model reuse.

Working Role

Working Role models roles in an organization, and specializes in **Project Manager**, **Software Engineering**, **Quality Manager**, etc.. A **Working Role** is responsible for one or more **Process Items**, or simply produces one or more of them. A **Person** plays one or more **Roles**, a **Role** is played by at least one **Person**.

3.2 Defining goals and measures

The next activity after defining the process model is the definition of measures. The well known, 'top down' Goal Question Metric approach (Basili and Rombach 1988), refined in standard IEEE1061 (1992) and in AMI (1992), is used.

The project goals were stated informally in 2. Now they are formalised in term of goals, subgoals and measures. In Table 1 we present a reduced version of the decomposition of goals relevant to process measures.

Table 1 Decomposition of goal in subgoals and measures

<i>Goal</i>	<i>Subgoal</i>	<i>Measure name</i>	<i>Measure definition</i> [<i>U of measure</i>]
Characterize the process	Characterise effort	effort per phase (ex. Analysis)	Σ Process_phase {Analysis}.works_mainly_on. Process_activity.duration [person hour]
		total effort	Σ Process_activity.duration [person hour]
	Characterise faults	faults per phase (ex. Analysis)	Cardinality(Process_phase {Analysis}.works_mainly_on. Anomaly_report) []
	Characterise reuse activities	reuse effort 1	Σ G++Library.study.duration + Σ G++Library.research /browsing.duration [person hour]
		reuse effort 2	Σ Class[derived from g++library = true].Process activity.duration [person hour]
		non reuse effort	Σ Class[derived from g++library = false].Process activity.duration [person hour]
	Characterise learning activities	oo learning effort	Σ G++.Study.duration + Σ C++.Study.duration [person hour]

Measures are defined in an OMT query language (see Morisio 1995). In this language 'Class' denotes all instances of **Class**, 'Class.attribute' selects all attributes of all instances of **Class**, 'Class.relation' selects all instances of the class linked to **Class** by the relation **relation**. 'Class[predicate]' selects all instances of **Class** that satisfy 'predicate'. 'Class{subclass}' is a particular predicate that selects all instances of **Class** that are also instances of **subclass**.

For instance in measure 'total effort' 'Process_activity' selects all instances of class **ProcessActivity**, 'Process_activity.duration' selects the attribute duration of such instances, ' Σ Process_activity.duration' adds the duration of all activities. The measure 'effort per analysis phase' sums the duration of all activities of any type executed on instances of **ProcessItem** on which the **Analysis** phase **worksMainlyOn**.

The measure 'reuse effort 1' sums the duration of all the **study** and **research/browsing** activities executed on instances of **G++Library**.

3.3 Technological support

In this activity procedures to collect, store and analyse measures are designed and the required tools are chosen. In this section we will describe only automated procedures and the tools chosen or developed to support them.

The automated procedures are: definition of the process model, definition of measures, definition of the database for measures, elaboration of raw data into measures. The manual procedures are: collection of process data (effort, faults) through paper time sheets and fault report sheets, insertion of process data into the database, definition of the algorithms to transform raw data into measures.

The tool chosen to support the automated procedures is G++, the same tool used by the measured development process. The advantages of this choice are discussed in section 4.2.

The definition of the process model uses the OMT editor of G++, the definition of measures uses the annotation feature of the editor.

The database for measures is generated from the process model using the C++ code generation facility of the tool and the library to add persistency to classes; part of the translation is done by hand and explained in more detail in the next section.

The elaboration of raw data into measures is made in terms of C++ function members belonging to the persistent classes. The definition of these function members is done by hand starting from the definition of measures on the process model and is explained in the next section.

3.4 Implementation

This section describes in more detail the C++ implementation of the automated part of the measurement process, basically the translation of the process model and process measures into a database, queries on it and forms to input raw data.

The Process Model Classes

Most of the classes in the process model were mapped directly to persistent C++ classes. Some of the sibling classes in the hierarchy have identical structure and behaviour; for instance, the subclasses of **ProcessActivity** only serve to distinguish between different types of activities. In these cases, the class hierarchy was simplified, and the classes in question were implemented as different instances of the parent class.

The queries need to identify some of the objects by a symbolic name. The classes for these objects therefore include a dictionary to map names to instances, along with a static (i.e. class) method returning an instance from a name.

Attributes are implemented as protected member variables which can be read through public member functions ("getters".) For the classes related to the measurement, the attribute values do not change after object creation, so they are simply passed to the constructor when the object is created.

As an example, the class declaration for **ProcessActivity** follows:

```
class ProcessActivity : public Object
{ protected:   Symbol name;
               Time duration;
               char type;
               Symbol means;

 public:       ProcessActivity(Symbol& name, Time t, char type, Symbol* means=0);
               Symbol& getName();
               Time& getDuration();
               char getType();
               Symbol& getMeans();
};
```

Relations

Relations were implemented using object pointers or collections of pointers, depending on the cardinalities involved. For instance, the relation between **ProcessItem** and **AnomalyReport**, where an **AnomalyReport** concerns one **ProcessItem**, while there can be many **AnomalyReports** for each **ProcessItem**, was implemented with a pointer to the involved **ProcessItem** object in the **AnomalyReport** objects and a collection of pointers to **AnomalyReports** in the **ProcessItem** objects.

For some relations, it is not worthwhile to maintain pointers for both roles. Typically, they are maintained only when they are needed for implementing a metric query.

A special kind of relation is the containment relation between a class and all its instances. Some of the queries need this to get to all instances of a specific class; other queries need to find a particular instance by its name without having prior visibility of that instance. When required, this was implemented by the above-mentioned dictionary of instances. For subclasses, instances must also be registered in the superclass' dictionary, so that queries accessing the superclass can find them. (This possibility is actually a beneficial side effect from using oo technology, difficult to implement with traditional databases.)

The declaration of **ProcessItem** is given below as an example of relation implementation.

```
class ProcessItem : public Object
{
protected:
    Symbol formalism;                // attributes
    String version;
    Symbol state;
    Symbol name;
    WorkingRole * responsible_role;  // relations
    WorkingRole * produced_by_role;
    ProcessPhase * produced_by_phase;
    OrderedCltn worked_on_by;
    OrderedCltn anomaly_reports;
    static Dictionary instance_dict; // map Symbol->instance pointer
public:
    ProcessItem(Symbol name);        // constructor/destructor
    ~ProcessItem();
    Symbol& getName();               // attribute access
    Symbol& getFormalism();
    void setFormalism(Symbol);
    String getVersion();
    void setVersion(String);
    Symbol getState();
    void setState(Symbol);
    OrderedCltn * workedOnBy();      // relation access
    OrderedCltn * anomalyReports();
    WorkingRole * responsibleRole();
    void responsibleRole(WorkingRole*);
    WorkingRole * producedByRole();
    void producedByRole(WorkingRole*);
    ProcessPhase * producedByPhase();
    void producedByPhase(ProcessPhase *);
    Dictionary& getInstanceDict();  // name->instance map access
    ProcessItem * getInstance(Symbol name);
};
```

Populating the Object Base

The database schema includes an explicit representation of the time sheet as a class, **TimeSheet**. This was done to simplify the population of the object base, since the time sheets must be entered manually. A time sheet object holds no attributes; it is simply a placeholder with pointers to the objects represented by the time sheet. The time sheet constructor takes the information from a line on a paper time sheet, creates a **ProcessActivity**, and sets up relations from the related objects. The **TimeSheet** object is filled with pointers to the objects involved, so that the **TimeSheets** can be browsed for verification against the paper sheets.

Queries

The G++ persistence framework does not include a query language, so the metrics queries had to be implemented manually. In the context of a larger project, this could surely be automated, but for the relatively small number of queries needed it was not found worthwhile to develop a query interpreter. In a larger project, however, it would be a useful addition, as it would allow experimenting with different queries to discover new metrics and to understand as early as possible which metrics are useful and which are not.

Queries are implemented by simply following the pointers that implement the relations. When following a link of cardinality N, the corresponding collection is traversed with a loop. Conditions, e.g. [type=group and means=e-mail], are translated into if statements that determine whether a link is followed further or not.

The innermost loop level performs the computation of the query's return value, e.g. by incrementing a counter for cardinality expressions or by accumulating a sum.

As an example, consider the metric "reuse effort 1", defined by the query $\Sigma_{G++Library.study.duration} + \Sigma_{G++Library.research/browsing.duration}$. To evaluate this query, we loop for each instance of the **GppLibrary** class, retrieving the collection of **ProcessActivity** pointers representing the relation **works on**. (**GppLibrary** inherits the collection needed to participate in this relation from its superclass **ExternalProduct**, see the process model in Figure 1.) Each **ProcessActivity** is queried for its name; if the name is "Study" or "Research/Browsing" the **ProcessActivity**'s duration is added to the sum.

The code that implements the query is given below.

```
// from ProcessActivity.h:      extern Symbol Study;
//                               extern Symbol ResearchBrowsing;

Time ReuseEffort1()
{
    Time sum = 0;
    Dictionary& gppLibInstances = GppLibrary::getInstanceDict();
    DO(gppLibInstances, GppLibrary, theGppLib)
        Collection& activities = *theGppLib->workedOnBy();
        DO(activities, ProcessActivity activity)
            Symbol& activityName = activity->getName();
            if (activityName == Study ||
                activityName == ResearchBrowsing) {
                sum += activity->getDuration();
            }
        OD
    OD
    return sum;
}
```

4 DISCUSSION

In the AEFTA project measures are based on a process model.

The first advantage in basing the definition of process measures on a process model is the clear definition of entities to be measured, in the actual context of the project under examination. Without the model, misinterpretations of definitions are very likely to happen, since the collection of process measures involves people and their judgment.

If the process model is defined in a formal or semi-formal language like OMT, analysis to produce definitions and their communication are greatly improved.

Moreover, object oriented concepts and technology have been applied to measurement, defining a process model in OMT, defining measures on it via an extension of OMT, using persistent C++ classes to implement the repository of measures, deriving semi-automatically the repository for measures from the process model, using an incremental/evolutionary lifecycle for the measurement process.

The advantages and rationales of these choices are now discussed.

4.1 Object oriented process model and evolutionary lifecycle

An evolutionary measurement process is a strict, important requirement since the software process, and the measurement process, change. This is fulfilled through careful use of characteristics.

Two categories of things can evolve in the measurement process: the process model (to adapt to changes in the real process, or to add detail) and measures (to adapt to new goals, or to follow the process model).

Impact of changes in the process model is reduced as follows. Whenever possible entities in the process are described as classes or associations: for instance actions are objectified in classes **ProcessPhase** and **ProcessActivity**. The process model is made mainly of the object model. The dynamic model is limited to some classes having rich or complex behaviour: for instance the why and when an anomaly report has to be issued. Moreover the dynamic model acts as documentation and is not used, up to date, in the definition of measures.

Changes in the definition of measures have two aspects: 1) changes in procedures for collection, computation and analysis, 2) compatibility with older measures. The former problem impacts only clerical rework and is reduced by the semi-automatic derivation of repository and function members. The latter problem is more dangerous since measures already taken and stored in the repository can become incoherent with the changed ones. The problem is tackled by exploiting generalisation/specialisation in defining a two level process model. One level, (abstract classes **Process Item**, **Process Activity** and the related abstract associations) is designed to be very robust against changes. The other level, or concrete level (specialisations of the above classes and the related specialised associations), is designed to change (by adding classes, by modifying associations) to accommodate evolutions. Measures are defined, whenever is possible, at the abstract level (let us call them abstract measures). Abstract measures are coherent, provided abstract classes and their relationships do not change, even if concrete classes change. On the contrary concrete measures can become incoherent if changes affect the concrete classes on which they are defined.

Any type of change in the process model and in measures is easily reflected in the supporting environment, since the repository is automatically generated from the process model.

4.2 Synergy between software process and measurement process

The measurement process uses the same concepts and technology as the software process. In particular the same development tool (G++) and language (C++) are used. In addition to the evident advantage of saving resources (same hardware, same tools, same training on tools and OMT method), a definite advantage is the sharing of the same mindset for programming and measuring. Process measures are collected mainly through manual filling of time sheets by the staff, and time sheets rely heavily on the process model for the definition of terms and relationships. Since the process model is described using OMT, a staff member can understand it using the same skill (s)he uses to understand project analysis and design documents. This skill improves with time and produces more reliable measures.

4.3 Measurement effort

The use of oo concepts and the definition of a process model impact heavily the distribution of effort for measurement in phases 1, 2, 3, 4 and 5 of the measurement process. The effort for the definition of measures (3) is very low, because the hard part of the job becomes the definition of the process model and of goals for the project (1 and 2). The technological phases (4 and 5) demand low effort too, partly because the know how is shared with the software process, partly because many components from the G++ library are reused.

When this paper was written the experiment was halfway from conclusion. Measures, and in particular measurement effort, will be published at the end of the experiment.

5 CONCLUSION

Object oriented concepts and technology have been applied to measure an object oriented project. This means defining a process model in OMT, defining measures on it via an extension of OMT, using persistent C++ classes to implement the repository of measures, deriving semi-automatically the repository for measures from the process model, using an incremental/evolutionary lifecycle for the measurement process.

The advantages of this approach are: sharing of tools, training, know-how and mindset with the rest of the project; possibility to define a process model and a set of measures which evolve easily; lower effort in the implementation of the technological support for measurement.

6 REFERENCES

- AEFTA consortium, (1994). Aefta project - Quality plan.
- AMI, Applications of Metrics in Industry (1992) - *ami Handbook, a quantitative approach to software management*, CSSE, South Bank Polytechnic, London.
- Bache R. , Bazzana G., (1994), *Software metrics for product assessment*, McGraw-Hill.

- Basili V.B., Weiss M.W. (1984), A methodology for collecting valid software engineering data, *IEEE Transactions on Software Engineering*, 10(6), Nov. 1984, 728-738.
- Basili V.B., Rombach H.D. (1988), The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Transactions on software engineering*, 14(6), June 88.
- Cook J. E., Wolf A.L., (1994), Towards Metrics for Process Validation, *3rd Int. Conf. on the Software Process*, Reston.
- Curtis B., Kellner M.I., Over J. (1992), Process modeling, *Comm. ACM*, Sept 92, 35(9).
- DEC, (1990), *BASEstar Open Introduction*.
- Feiler P.H., Humphrey W.S., (1993), Software Process Development and Enactment: Concepts and Definitions, *2nd Int. Conf. on the Software Process*, Berlin.
- Fenton N., (1991), *Software Metrics - A Rigorous Approach*, Chapman & Hall.
- Hetzel B., (1993), *Making software measurement work*, QED.
- IEEE, (1992), IEEE 1061 Standard for a Software Quality Metrics Methodology.
- Jones C. (1991), *Applied software measurement*, Mc Graw Hill.
- Matsumoto K., Kusumoto S., Kikuno T., Torii K., (1993), A new Framework of Measuring Software Development Processes, *Proceedings of the 1st Int. IEEE Software Metrics Symposium*, Baltimore, MD, 21-22 may 1993.
- Menga G., Elia G., Mancin M., (1993), G++: an environment for object oriented design and prototyping of manufacturing systems, In *W. Gruver, G. Boudreaux, ed., Intelligent Manufacturing: Programming Environments for CIM*, Springer Verlag, 1993.
- Morisio M., (1995), A Methodology to Measure the Software Process, *Proceedings of the Annual Oregon Workshop on Software Metrics*, Portland, OR, 5-7 june 1995.
- Osterweil L., (1987), Software Processes are Software Too, *Proc. 9th ICSE*, April 1987.
- Pfleeger S.L., McGowan C., (1990), Software Metrics in the Process Maturity Framework, *Journal of Systems and Software*, n. 12, 1990, 255-261.
- Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenzen W., (1991), *Object oriented modelling and design*, Addison Wesley.
- Roche J., Jackson M., Shepperd M., (1994), Software Measurement Methods: an Evaluation and Perspective, *3rd Symposium on Assessment of Quality of Software Development Tools*, Washington, June 1994.
- Wolf A.L., Rosenbaum D.S., (1993), A Study in Software Process Data Capture and Analysis, *2nd Int. Conf. on the Software Process*, Berlin, 1993.

7 BIOGRAPHY

Amund Aarsten is a Ph.D. student at Politecnico di Torino, writing his thesis on concurrent and distributed object-oriented systems. His research interests lie in the fields of design patterns, programming languages, and software process modeling.

Maurizio Morisio is a researcher at Politecnico di Torino. His research interests include object oriented software metrics, CASE tools evaluation and selection, software process modelling, requirements specification and design formalisms. He was in charge of measuring the AEFTA project. He received a Ph.D. in software engineering from Politecnico di Torino in 1989.