2nd SC@RUG 2005 proceedings

Smedinga, Rein; Terlouw, Jan

Proceedings Student Colloquium 2004-2005

Rein Smedinga,
Jan Terlouw

Computing Science
University of Groningen

Studcoll 2005 proceedings

Rein Smedinga
Jan Terlouw
editors

2005
Groningen

ISBN 90-367-2326-4
Publisher: Bibliotheek der R.U.
Title: Proceedings 2nd Student Colloquium 2004-2005
Computing Science, University of Groningen
NUR-code: 980

Contents

1. About StudColl

Introduction StudColl is a course that master students in computing science follow in the first year of their master study at the University of Groningen.

In the academic year 2004-2005 StudColl was organized for the second time as a conference. Students wrote a paper, participated in the review process, gave a presentation and were session chair during the conference.

The organizers Rein Smedinga and Jan Terlouw would like to thank all colleagues, who cooperated in this StudColl by collecting sets of papers to be used by the students and by being expert reviewers during the review process. They would also like to thank Marjolein van der Werff from the Faculty of Arts for her help in organizing this course.

In these proceedings all accepted papers are published.

Organizational matters StudColl 2005 was organized as follows. Students were expected to work in teams, consisting of two persons. The student teams could choose between different sets of papers, that were made available through *Nestor*, the digital learning environment of the university. Each set of papers consisted of three papers about the same subject (within Computing Science). Soms sets of papers contained conflicting meanings. Students were instructed to write a survey paper about this subject including the different approaches in the given papers. The paper should compare the theory in each of the papers in the set and include own conclusions about the subject.
Some teams proposed an own subject.
Six teams previously followed the course "Research Methodologies" and could use StudColl2004 for presenting their results, both in a paper and by giving a presentation.

After submission of the papers individual students were assigned one paper to review using a standard review form (see Appendix A of the previous StudColl2004 proceedings). The colleagues who had provided the set of papers were also asked to fill in such a form. Thus, each paper was reviewed three times. Each review form was made available to the authors of the paper through *Nestor*.

All papers could be rewritten and resubmitted, independent of the conclusions from the review. After resubmission each reviewer was asked to rereview the same paper and to conclude whether the paper had improved. Rereviewers could accept or reject a paper. All accepted papers can be found in these proceedings.

All students were asked to present their paper at the conference and act as a chair or as discussion leader during one of the other presentations. The audience graded both the presentation and the chairing or leading the discussion. Marjolein van der Werff of the Faculty of Arts gave an introductory lecture about general aspects of presentation techniques to help the students with their presentation.

Students were graded both on the writing process, the review process and the presentation. Writing and rewriting counted for 40% (here we used the grades given by the reviewers and the rereviewers), the review process itself for 15% and the presentation for 45% (including 5% for the grading of being a chair or discussion leader during the conference). For the grading of the presentations we used a selected number of judgements from the audience and calculated the average of these.

On January 31st and February 1st 2005, the actual conference took place. Of each writing team both authors presented half of the presentation. Both days, we had ten presentations, each consisting of a total of 30 minutes for the presentation and 10 minutes for discussion. As mentioned before each presenter also had to act as a chair or as discussion leader for another presentation during that day. The audience was asked to fill in a questionnaire and grade the presentations, the chairing and leading the discussion.

A Comparison of Connected Filters

Roland Veen   Bjørn Lindeijer

{r.j.veen, t.lindeijer}@wing.rug.nl

Abstract

Some algorithms and operators related to Connected Filtering are discussed. This includes an elaborate discussion about the *Max-Tree* approach, a discussion about the Fast Level Lines Transform and we briefly mention the union-find method.

Keywords — Connected set operators, attribute filters, union-find, max-tree, fast level lines transform, connectivity.

1 Introduction

Connected set operators are often used when preservation of shape is very important in image filtering, for example in medical applications. These operators have evolved from working on binary images to handling complex grey-scale images, which makes efficiency a very important aspect in which a lot of research is done. In this paper we present an overview of a number of algorithms to extract connected components from images, and Connected Set Operators that can perform filtering on these connected sets depending on their properties. The goal is to assert the applicability and usefulness of these operators and related algorithms.

In the next section we will briefly summarize some of the aspects of connected set theory. In the then following sections, algorithms for building structures of connected components are discussed, and then some different connected set operators are described.

2 Theory

For clarity, we briefly summarize some important aspects about Connected Set Operators.

Instead of operating on individual pixels, connected set operators operate on flat zones of images. The image is divided into disjoint sets of pixels, which form the connected shapes.

In figure 1, you can see an example of a basic binary structural and area opening and closing. It is important to notice that these operators do not introduce new edges in the image, thus preserve the shape of the components. This is specifically important in medical applications, where introduction of artifacts is not tolerated.

3 Algorithms for building connected set representations

In this section we will give an overview of some of the methods used and in use for the construction of connected set representations.

3.1 Pixel-Queue

3.1.1 Description

The Pixel-Queue algorithm was briefly mentioned in [1]. The algorithm, not very surprising, scans an image using a pixel queue to create a list of all regional maxima. Then these maxima are put in a priority queue based on their grey-level. These maxima are then processed sequentially, essentially flooding the region surrounding it with the same grey-level until either a pixel is encountered with a higher grey-level, or the covered area equals a predefined size. This results in a grey-level map which can be used to apply connected set operators on the fly.

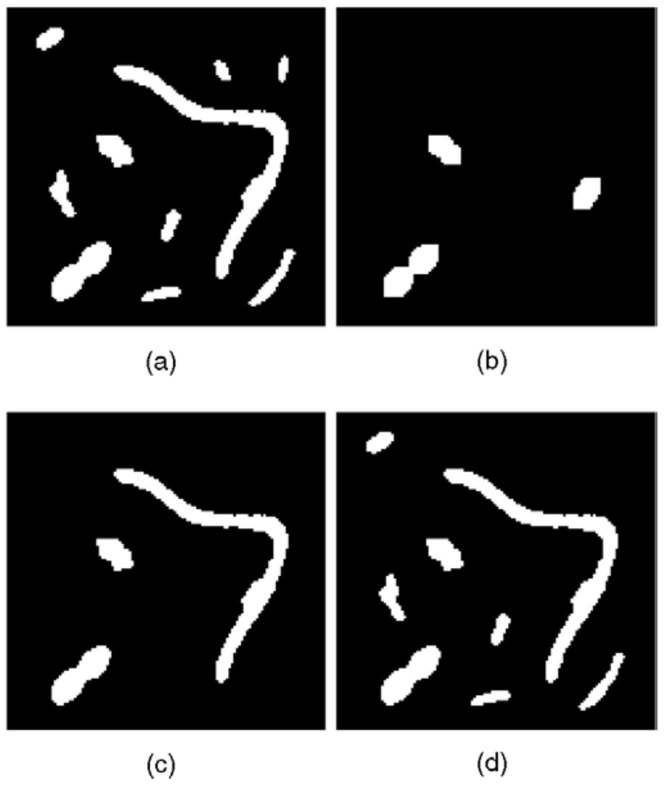


Figure 1: Area versus structural openings: (a) An original image, (b) Structural opening with 7×7 structuring element, (c) opening by reconstruction using same element, (d) Area opening with $\lambda = 49$. the shapes of connected foreground components remain unchanged. Source: [1]

3.1.2 Evaluation

The application of operators on the fly potentially allows for reducing computation time by stopping when criteria are met or have no chance of being met anymore. However this property is also one of the drawbacks of the pixel-queue algorithm, since the deconstruction of the image in components is not reusable, and has to be redone for every change in parameters. The *Max-Tree* approach, described next, separates the application of the connected operator from the deconstruction of the image.

3.2 Max-Tree

3.2.1 Description

The Max-Tree was introduced by Salembier et al. [3]. To describe a *Max-Tree* we first picture an image as a height map, for example with the bright areas higher than the dark areas. The peaks in this image form the leaves of the *Max-Tree* and the lowest level is the root. Each node in the tree corresponds with a connected component in a slice through the landscape at a certain height. All nodes except for the root node point to their parent, which is the component containing them that is lower. When the parent of a node happens to be exactly the same shape, it can be removed to simplify the tree.

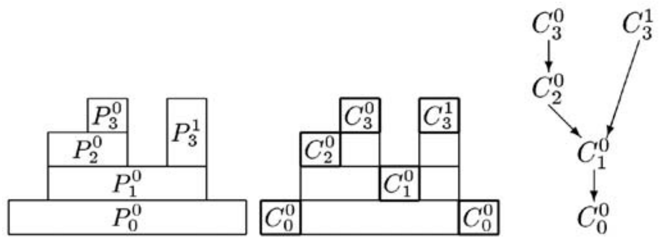


Figure 2: Visualization of a Max-tree structure in one dimension. Source: [1]

In figure 2 an example is given of a *Max-Tree* structure generated from a 1-dimensional image. The leftmost picture shows the slicing of the height map and already superfluous nodes have been removed. The middle image shows the component assignments to the actual pixels. Finally the rightmost image shows the *Max-Tree* itself.

In summary the *Max-Tree* gives us a structural tree presentation of the image. This presentation is useful for processing the connected components as we'll show later. First, we will briefly describe the construction of a a *Max-Tree*.

3.2.2 Building a Max-Tree

The *Max-Tree* construction can be described as recursive application of binarization using a certain increasing grey-level threshold on the image and connected component analysis on each resulting binary image. The final result is that each pixel will be tagged with the connected component node in the tree it belongs to, and the tree itself. The growing process of the *Max-Tree* is shown in figure 4.

Salembier et al. [3] describes a very fast algorithm to construct a *Max-Tree* using recursive flooding in combination with a *first-in-first-out* (FIFO) pixel queue. Its execution time it typically of less than one second on a Sun-Sparc 10 for 256x256 images of 256 grey-levels. The complexity of this algorithm is $O(N)$ because the floodfill, pruning and producing the output are all of linear complexity.

3.2.3 Filtering a Max-Tree

Given a *Max-Tree* and a filter criteria on connected components, there are several ways to go about filtering out the components. When the filter criteria is increasing towards the root of the *Max-Tree* the obvious thing to do is to remove all nodes that do not reach the criteria threshold, and their children. This is for example the case for an area opening. The components in a *Max-Tree* are naturally decreasing in area towards the leaves.

However, many filters are not necessarily increasing. Examples are filters on shape, entropy or motion. In this case a path from the root of a *Max-Tree* towards one of the leaves will contain an arbitrary amount of nodes not reaching the criteria threshold spread arbitrarily.

There are three straightforward ways to determine which parts of such a path should remain in the tree, illustrated in figure 3. The “Min” decision chooses to preserve only the nodes until the threshold isn’t reached for the first time. The “Max” decision chooses to preserve all nodes up to the last time that the threshold has been reached. Finally the “Direct” decision preserves any nodes that reach the threshold and removes all others.

While these decision rules are very straightforward, the results are often not satisfactory. The ”Min”decision will quickly leave out too much, whereas the ”Max”decision leaves in too much. The problem here is that the decisions are local and do not depend on the decision of neighboring nodes, making the approach not robust in practice. This is also the case for the ”Direct”decision. A solution to this is mentioned when we describe anti-extensive operators in section 4.2.1.

3.3 Fast Level Lines Transform

The Fast Level Lines Transform, which is presented in [2], builds a representation of the image in the form of a tree containing shapes in the image. It uses both a Max-Tree and a Min-Tree to create an inclusion map of the shapes. The advantage of having such an inclusion map is the ability to filter out small shapes by pruning the tree at a certain

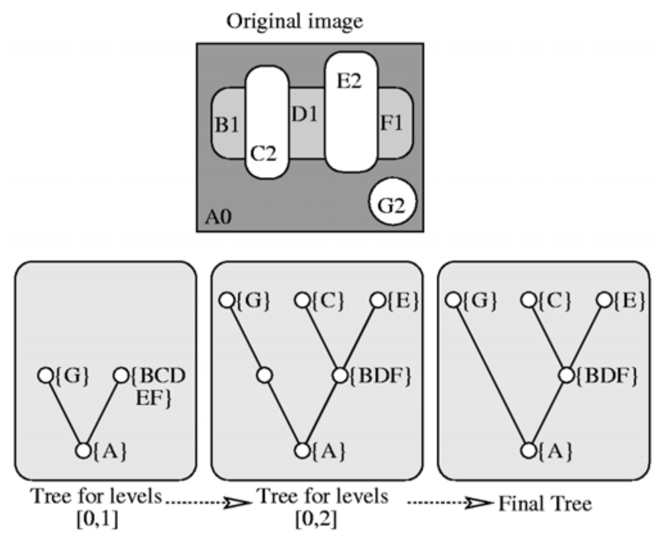


Figure 4: Example of Max-Tree construction. Source: [3]

depth, which is illustrated in figures 5 and 6. When looking at the decomposition of the image with respectively Max-Tree, Min-Tree and FLLT, we see this difference in which shapes are identified in an image.

To obtain the tree of shapes, thresholding of the image at certain grey-levels can be performed, which is very costly. However, when taking advantage of the tree structure, FLLT builds and merges upper and lower level sets as computed by a Max-Tree and Min-Tree algorithm, which results in a drastic speed improvement. A nice example of the FLLT is shown in figure 7.

3.4 Union-find method

3.4.1 Description

The union-find method [1] does essentially the same as the *Max-Tree*. The pixels are processed in grey-level order, in combination with Tarjan’s union-find algorithm for keeping track of disjoint sets [4]. The algorithm uses a tree-based approach where two objects x and y are members of the same set if and only if their they are nodes in the same tree. Sets can be merged based on properties of two nodes r and p, which is done by setting a common root. The end result is a tree like the *Max-Tree*.

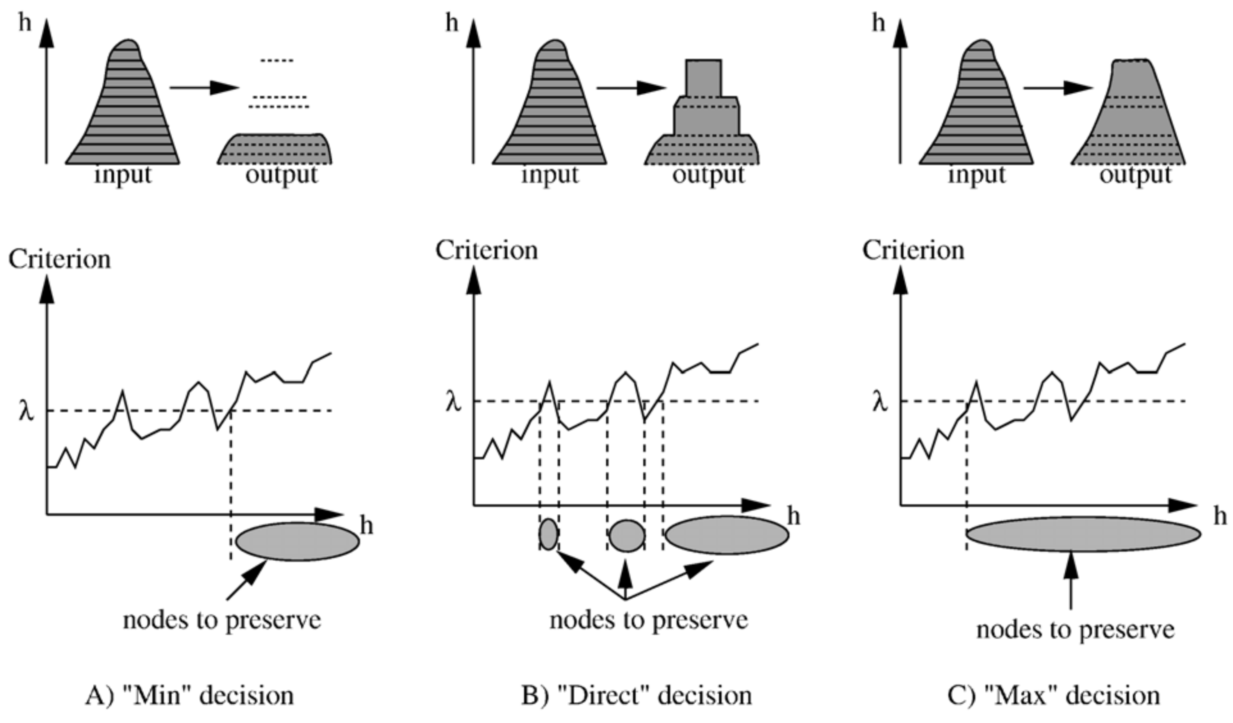


Figure 3: Illustration of various decision rules in the case of non-increasing criterion. Source: [3]

3.4.2 Evaluation

The advantage of the union-find method is that it doesn't walk through the image level by level, but instead it can process the image in parallel while keeping track of connected components found. Another advantage of the union-find method is its low memory usage. On large data sets this translates to major speed increases.

4 Connected Set Operators

A few connected set operators are discussed.

4.1 Scale-invariant thinning operator

One example that uses the union-find method is shown in figure 8. This example uses a 256^3 volumetric dataset, showing that this method can perform well even with large datasets because of its limited memory usage. The criterion used is based on the momentum of the shape, distinguishing long narrow objects from more compact ones, effectively filtering out noise and preserving the blood vessels.

4.2 Some Anti-Extensive Operators

In Salembier et al. [3] the flexibility of the *Max-Tree* is shown using many examples. We'll mention them here.

4.2.1 Simplicity criteria

The simplicity of a connected component is given by the ratio between its perimeter P and area A.

$$Simplicity(C) = A(C)/P(C)$$

This criterium is not necessarily increasing, hence a Viterbi algorithm is used to find the shortest path from root to leaf. The cost of this path is then thresholded. This takes into account much better the overall appropriateness to include a connected component than the Min, Max or Direct decision methods would.

4.2.2 Motion

Given a *Max-Tree* of a single image from an animation, the decision of which components to be included can be made by comparing the components with their contents in previous time-steps. This is

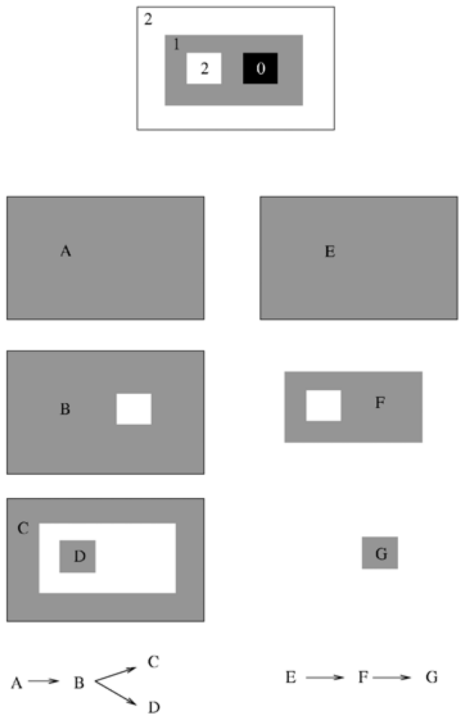


Figure 5: Connected components of upper and lower level sets. Source: [2]

also not necessarily an increasing criteria, solved in a similar way as the previous example.

5 Conclusions

The discussed algorithms for performing image operations with connected sets look very interesting with respect to flexibility and performance. The FLLT looks very useful, and of the operators, the scale invariant attribute thinning has already showing a lot of promise in medical applications. When comparing the union-find method to the *Max-Tree*, the former seems to be better in practice (see figure 8), since it is optimized for memory, which becomes more apparent when using real-size images.

Acknowledgements

Special thanks to Dr M.H.F. Wilkinson, Dr. J. Terlouw and Dr. R. Smedinga.

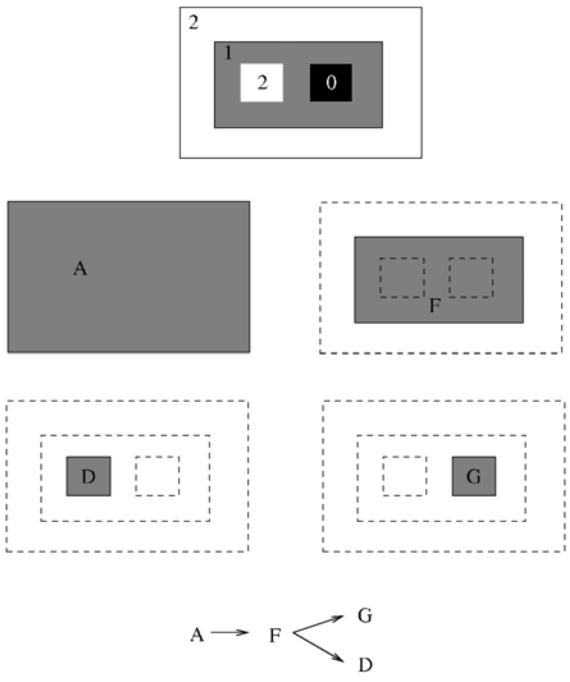


Figure 6: The tree given by the FLLT. Source: [2]

References

[1] A. Meijster and M. Wilkinson. A comparison of algorithms for connected set openings and closings, 2002.

[2] P. Monasse and F. Guichard. Fast computation of a contrast invariant image representation, 1998.

[3] P. Salembier, A. Oliveras, and L. Garrido. Anti-extensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.

[4] R.E. Tarjan. Efficiency of a good but not linear set union algorithm, 1975.

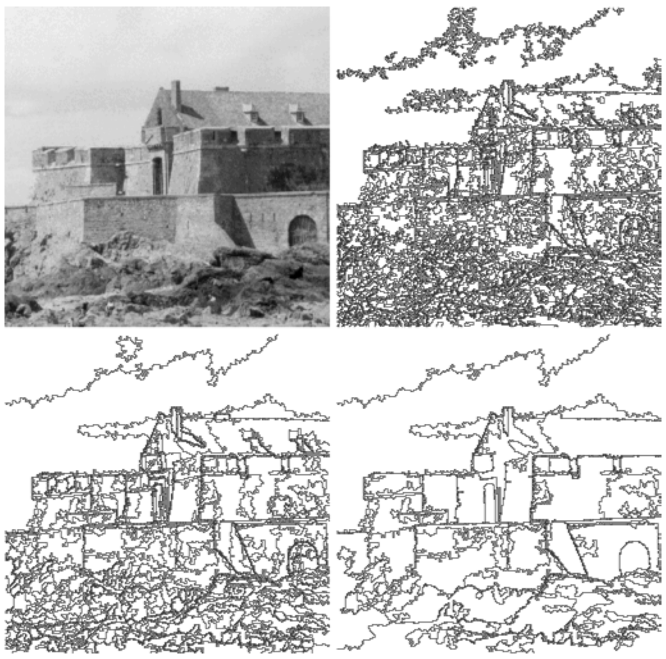


Figure 7: Example of Fast Level Lines Transform. Shown are the boundaries of shapes of respectively 10, 40 and 800 pixels. Source: [2]

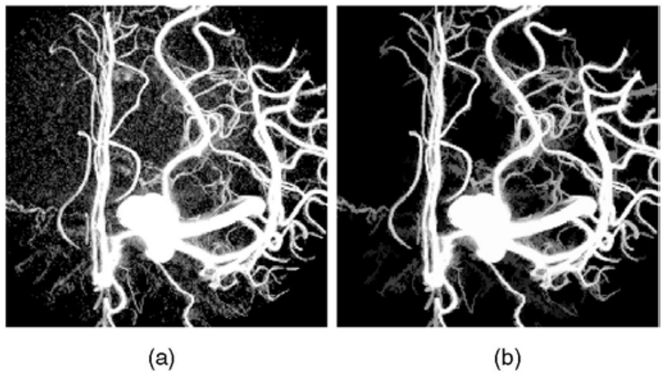


Figure 8: A medical application of the scale invariant attribute thinning. Source: [1]

Assembling Classes at Runtime

By Peter Hut (p.h.hut@student.rug.nl) and Zef Hemel (z.hemel@student.rug.nl)

Abstract: This paper will describe the problem of assembling classes at runtime. Four solutions will be considered and their advantages and disadvantages explored. The solutions are: the Type Object pattern, the Adaptive Object-Model, the UML Virtual Machine and using a Dynamic Language.

1. The problem

Dynamic and configurable systems are the upcoming trend. Most systems are demanded to be flexible and easily extensible. This is needed to adapt the system to changing business rules (Rouvellou, 1998). An example of this is a system designed for administration of measurements done in a hospital. There are many different possible measurements. It is not feasible to design a subclass for each type of measurement. Furthermore it is very likely that new types of measurements need to be added to the system after deployment. The adding of new types of measurement should be possible for non-programmers. Additionally, the system has to keep running at all times. So shutting down the system, adding the new classes, recompiling the system and then starting it up again is not an option. The solution for this problem is assembling classes at runtime.

As in this example, there are three main reasons to need runtime assembling of classes. They are:

- The number of subclasses is unknown upfront
- The number of subclasses is huge
- Changes to the system have to be made without the system going down

2. Four solutions

For these problems four possible solutions exist:

- The Type Object Pattern
- Adaptive Object-Models
- The UML Virtual Machine
- Using a Dynamic Language

Each of these solutions will be discussed separately and at the end the solutions will be compared to one another.

2.1. The Type Object Pattern

The Type Object Pattern is a simple way to assemble classes at runtime (Johnson, 1998). An example will describe how it works.

A library contains a lot of different books. It also has multiple copies of one book. Some information about such a book, like whether it has been borrowed and by whom, is different for each copy. So therefore, an instance of the Book class is necessary for each copy. This leads to duplication of information.

A possible solution is to create a class for each title (for example 1984Book, TomSawyerBook and TheTimeMachineBook). In the class itself common data for all copies is stored; each instance represents a copy and stores data such as who has borrowed it.

This solves the data duplication problem, but is not very elegant. Another problem is that for every single title a subclass has to be created. This is not only a lot of work, but would also mean that for every addition of a title a new class has to be written.

The Type Object pattern solution suggests creating two classes: Title and Book. The Title class would contain data common to all copies of that title and the Book class contains a reference to the Title class and stores data specific to that particular copy (such as who borrowed it).

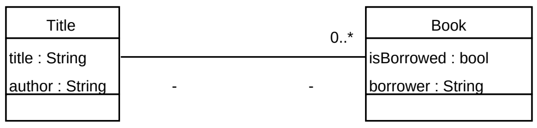


Figure 1 – Type Object pattern example.

Now, when a new copy of a book comes in, a new instance of the Book class can be created that references the Title it belongs to. If a new title comes in all that has to be done is the creation of an instance of the Title class and an instance of the Book class for each copy that came in.

Figure 2 gives the more general structure of the Type Object pattern. The Type Object pattern has two concrete classes, one that represents objects and another that represents their type. Each Object-class instance has a pointer to its corresponding TypeObject object.

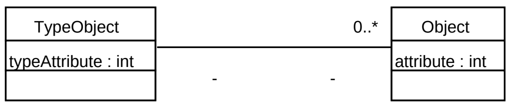


Figure 2 - The Type Object pattern.

2.2. Adaptive Object-Models

If it is not only necessary to create new classes at runtime, but also to customise the attributes, associations and the behaviour of these classes, then the Type Object pattern isn't fully suitable. In the library example no new attributes have to be defined, these are all known upfront, but in some cases the attributes that the objects will have is unknown. In many cases it is then possible to use an Adaptive Object-Model.

To make the customisation of attributes possible, the Type Object pattern is combined with the Property pattern (Foote, 1998). The property pattern makes it is possible to dynamically add attributes to classes. The two patterns combined lead to the architecture as represented by Figure 3.

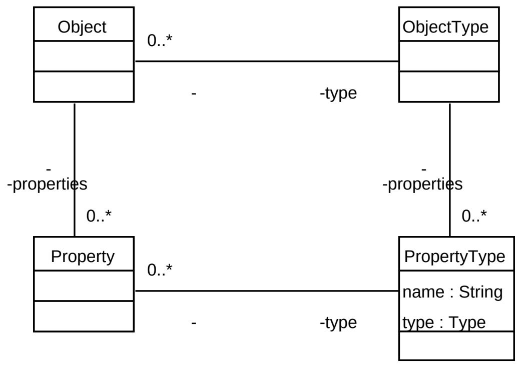


Figure 3 – Type Object and Property pattern combined.

The ObjectType object needs to store a list of properties (and their types) that its instances will have. And each Object needs to store a list of values for these properties.

To handle relationships between objects two subclasses of PropertyType are introduced: AssociationType and AttributeType. And two subclasses of Property: Association and Attribute. See Figure 4 - Handling associations (Yoder, 2003).

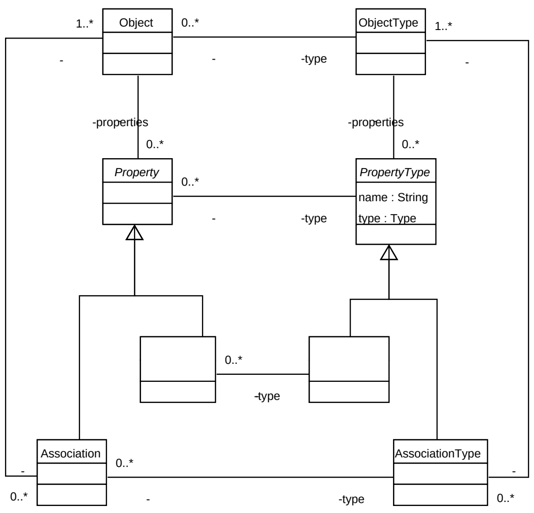


Figure 4 - Handling associations (Yoder, 2003)

To define the behaviour of an object, the Strategy pattern can be used. Strategies can, for example, be used to validate the values of properties.

In general a Strategy is an object that represents an algorithm. The Strategy pattern defines a standard interface for a family of algorithms so that clients can work with any of them. If an object's behaviour is defined by one or more strategies then that behaviour is easy to change. (Yoder, 2001)

Figure 5 is a UML diagram of applying the Type Object pattern twice with the Property pattern and then adding Strategies (called Rules in the diagram).

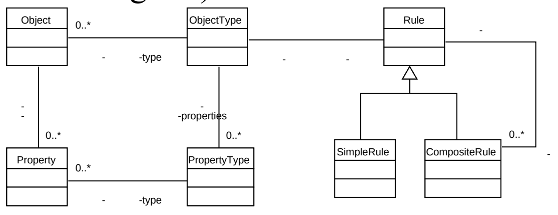


Figure 5 - Handling strategies.

To show how Adaptive Object-Models can be applied, an example will be used from (Yoder, 2001). In a hospital, measurements are done on people. The number of different kinds of measurements is very large and not known upfront. That’s why it should be possible for hospital personnel (not just programmers) to define new kinds of measurements. Each measurement has different properties and relations.

To implement these requirements, the Adaptive Object-Model is used. This results in the class diagram in Figure 6. There are two types of Observations. One type consists of discrete values such as blood type or eye colour (Trait). The other has continuous values such as weight or height (Measurement).

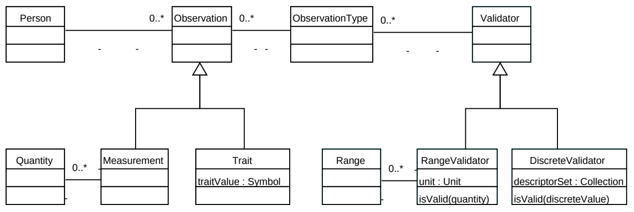


Figure 6 - Hospital example.

To create a new kind of observation a new instance of ObservationType has to be created. To be able to validate the values of the measurements, the Strategy Pattern is applied. The validators can be used by the ObservationType to check whether the value of an Observation is legal. Because of the two kinds of observations, for Measurements and traits, there are two kinds of validators. One represents a range of Measurements and one represents a set of Traits. A validator is just an algorithm telling whether a value is valid.

2.3. The UML Virtual Machine

The UML Virtual Machine is a totally different approach to software development, based on the Model Driven Architecture (MDA). MDA encourages efficient use of system models in the software development process and it supports reuse of best practices when creating applications. The main idea is to develop as much as possible of the application through design and less through implementation.

UML is a well-known standard for designing and specifying software. It knows different diagrams, such as the class diagram to define the structure of the software and other diagrams to specify the behaviour of the software. The most used diagrams for behaviour are state chart diagrams and collaboration diagrams. In version 1.5 of UML (OMG, 2003), action semantics were added to allow full specification of the behaviour of software. Additionally the Object Constraint Language (OCL) can be used to define constrains on objects.

It's very common to first design an application in a UML application (like Visio or Rational Rose) and then implement it in some programming language. Sometimes the UML diagrams are used to automatically generate some of the code, interfaces and stubs for the classes and methods.

Executable UML (Mellor, 2002) takes this a step further. It eliminates the implementation phase. The Executable UML application will read the UML specification and compile it to executable code. This, however, does still mean that the system has to be restarted each time the object model is adapted. So it can't assemble classes at runtime.

UML Virtual Machines (Riehle, 2001) on the other hand are different. UML Virtual Machines will read the UML specification and interpret it on the fly. While the application is running, the UML specification can be changed. New classes, attributes and associations can be added and behaviour can be defined, all without shutting down the application. Algorithmic detail can be added as hand-programmed policy classes that fit into a well-defined extension architecture.

The UML Virtual Machine, in contrast to the Type Object pattern or Adaptive Object-Models, is not a set of patterns that can be used, but rather a runtime environment that runs the application. It's a product that can be bought. At this moment, though, there is no working implementation of the UML Virtual Machine. (Riehle, 2001) is working on one, but it's only slowly progressing. There's also a group at the university of Massachusetts (Lall, 2004) working on an implementation (expecting to deliver in May, 2005).

This system could be used as a solution to assemble classes at runtime, as the UML specification can be changed at runtime, including defining new classes. The problem is that a UML tool will have to be used to design the new classes, which not every layman will understand. On the other hand, it is imaginable that an application can be created that interfaces with the UML Virtual Machine and exposes a user-friendly GUI to the user to define new classes, associations and behaviours.

Unfortunately there is no working implementation of the UML Virtual Machine yet. Therefore only guesses can be made about its opportunities.

2.4. Dynamic Languages

Dynamic Languages such as Python and Ruby allow runtime assembling of classes and at runtime adaptation of classes and objects, by design; it is part of the dynamic nature of the languages (Mertz, 2003). Everything that can be done at "compile" time can be done at runtime.

It is possible to generate classes and methods; and attach new properties and methods to classes and individual objects. To show this, a simple Python example is included in Listing 1 at the end of this paper. The example implements a couple of elements of the hospital example as presented in section 2.2.

3. Comparing the solutions

The table on the next page gives a quick overview of the differences between the discussed solutions. In the next sections we'll discuss what each row means.

3.1. Design complexity

Design in this context means how much the complexity of the design of your application increases because of adding class assembling features.

Before the Type Object pattern or an Adaptive Object-Model can be used effectively it is necessary to understand the principles of the chosen method. These are not always obvious. In practice both will result in more design complexity, in particular Adaptive Object-Models, which requires quite a complex class structure.

As an application can be designed for a UML Virtual Machines as usual and the changing of classes at runtime comes with using the UML Virtual Machine, the design complexity is low.

Because class assemblage is very common and natural in a Dynamic Language, no special design tricks have to be applied. Therefore the design complexity of adding features to assemble classes at runtime is low, but the availability of these features might not be obvious from the design.

3.2. Implementation complexity

When the Adaptive Object-Model is used it will result in a more difficult architecture than when only the Type Object pattern is used. Consequently the first will be harder to implement.

When it is possible to use the UML Virtual Machine as a component off the shelf, not much implementation work is needed, except maybe a graphical user interface to allow laymen to create new structures. But of course, this all depends on the implementation of the UML Virtual Machine, which does not exist as of yet. If an entire UML Virtual Machine has to be implemented, the implementation complexity would become very high.

To use a Dynamic Language, familiarity with the language is necessary and a Dynamic-Language interpreter has to be used or integrated in the application. Implementing the assembling of classes at runtime is very simple.

3.3. Implementation language constraints

For the Type Object, Adaptive Object-Model and UML Virtual Machine no particular language features are required. For the Dynamic Language solution to be used, a Dynamic Language is necessary (obviously).

3.4. Application-embedded domain knowledge

When the Type Object pattern is used the classes that can be assembled at runtime can only be changed in minimal ways and are, for the most part, designed with the domain knowledge in mind. Therefore a lot of domain knowledge is embedded in the application.

When an Abstract Object-Model or Dynamic Language is used, even the attributes of the classes can be changed at runtime. This means the system can still be adapted to a domain in which it is used, even while it is running. Therefore when designing a system using an Abstract Object-Model or Dynamic Language, only limited domain knowledge has to be embedded in the application, as a lot can be defined at runtime. This also depends on the tools the system has available to change or add classes at runtime. In most cases the purpose of using an Abstract Object-Model would be to let the user adapt the system to domain in which it is used.

The UML Virtual Machine, like Dynamic Languages, allows you to assemble any kind of class at runtime as desired, with any kind of behaviour desired. Therefore very little domain knowledge has to be embedded into the application upfront.

3.5. Change properties/association at runtime

With an Abstract Object-Model, UML Virtual Machine and Dynamic Language it is possible to add and remove properties of a class. With just the Type Object pattern it is not.

3.6. Change behaviour at runtime

The Type Object pattern does not allow you to change behaviour at runtime, for an Adaptive Object-Model the behaviour can only be changed by using the Strategy pattern (see Figure 5). The user, in many cases, can choose one of pre-defined strategies which is somewhat limiting, but enough in many cases.

Depending on the chosen way to implement behaviour in the UML Virtual Machine, the system's behaviour can be fully changed at runtime. In a Dynamic Language it is possible to generate code at runtime and instantly interpret it. So any kind of logic can be generated at runtime.

3.7. Flexibility

Using the Type Object pattern means that the new classes that can be generated will have pre-defined properties and behaviour.

When an Abstract Object-Model is used the properties of the (at runtime assembled) classes can be modified and behaviour can be chosen from pre-defined strategies.

The UML Virtual Machine allows the creation of whole new programs and there are no constraints. The same goes for the Dynamic Language approach.

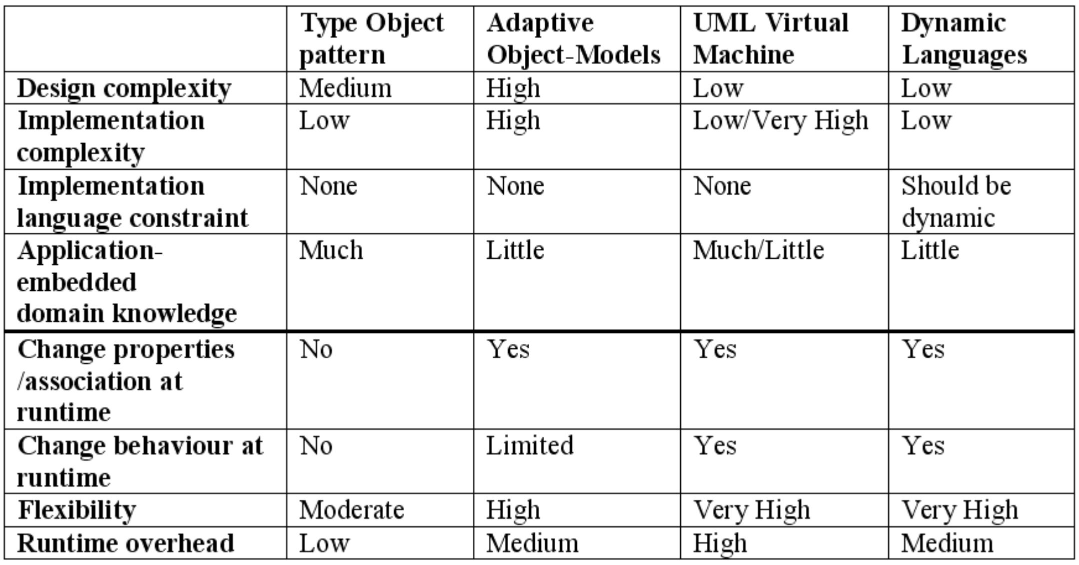
3.8. Runtime overhead

If the Type Object pattern is used instead of a normal class-subclass relationship it would mean slightly more runtime overhead than usual, as the objects need to keep track of their TypeObject-relation themselves. Also some requests to the object might need to be forwarded to its TypeObject.

If, on the other hand an Adaptive Object-Model would be used it, would give more overhead than the Type Object pattern, as it keeps track of properties and relationships as objects.

When the UML Virtual Machine is used, the object model is re-implemented on the object model of the implementation language. On top of that the application is run. This will of course mean a lot more overhead compared to implementing the application directly in the implementation language.

For Dynamic Languages, the runtime overhead is hard to determine. It highly depends on the implementation of the Dynamic Language's interpreter. Most implementations work like the Adaptive Object-Models, others emit machine code at runtime for the at-



	Type Object pattern	Adaptive Object-Models	UML Virtual Machine	Dynamic Languages
Design complexity	Medium	High	Low	Low
Implementation complexity	Low	High	Low/Very High	Low
Implementation language constraint	None	None	None	Should be dynamic
Application-embedded domain knowledge	Much	Little	Much/Little	Little
Change properties /association at runtime	No	Yes	Yes	Yes
Change behaviour at runtime	No	Limited	Yes	Yes
Flexibility	Moderate	High	Very High	Very High
Runtime overhead	Low	Medium	High	Medium

runtime generated classes. In that case there is hardly any runtime overhead.

4. Conclusion

If a system needs to be created in which:

- there are a large amount of sub-classes of one class;
- the number of sub-classes upfront is not known; or
- it is necessary to make changes without the system going down;

then a solution can be used that allows the assembling or change of classes at runtime. Four solutions and their advantages and disadvantages were discussed.

The Type Object pattern is quite simple and can be used in any design but offers limited flexibility.

The Adaptive Object-Model is more flexible and allows the system to be adapted later to better fit organizational changes. This means less domain knowledge is embedded into the application compared to a normal situation or when the Type Object pattern is used. The drawback of using an Adaptive Object-Model is that the system will be more difficult to understand.

An entirely different solution is to use a UML Virtual Machine. The design of the system will be the same as usual, but it would be possible to also give the user the option to add and change classes at runtime. From this the same advantage is obtained as from an Adaptive Object-Models, but not the disadvantage that the design is more complex than usual. The problem is that currently no finished implementation is available of a UML Virtual Machine and implementing one would mean quite an investment.

The last solution that was discussed, Dynamic Languages, allows for a normal design and implement of a system, and also allows the addition of features to allow users to make runtime changes to the classes. The disadvantage in this case is the restriction to a limited group of implementation languages.

5. References

Foote, B., Yoder, Y.W., “Metadata and Active Object Models”, Proceedings of Plop98. Techincal Report #wucs-98-25. Washington University Department of Computer Science. URL: http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P59.pdf (1998)

Johnson, R., Wolf, B., “Type Object”, Pattern Languages of Program Design 3, Addison Wesley. (1998)

Lall, A., Malinowski, A., Qureshi, M., Solaiappan, K., "UML Virtual Machine", URL: http://umlvm.cs.umb.edu (2004)

Mellor, S.J., Balcer, M.J., "Executable UML: A Foundation for Model Driven Architecture", Addison-Wesley Pub Co; 1st edition, ISBN 0201748045 (2002)

Object Management Group Inc., "Unified Modeling Language specification 1.5" (2003)

Mertz, D., “A Primer on Python Metaclass Programming”, URL: http://www.onlamp.com/pub/a/python/2003/04/17/metaclasses.html (2003)

Riehle, D., Fraleigh S., Bucka-Lassen, D., and Omorogbe, N.,”The Architecture Of A UML virtual machine”, Proceedings of OOPSLA'01. ACM Press, New York, 327-341. (2001)

Rouvellou, I., Degenaro, L., Rasmus, K., Ehnebuske, D., McKee, B., “Extending business objects with business rules”. Proceedings on Software Engineering: Education & Practice. Page(s): 238 - 249. (1998)

Yoder, J. W., Johnson, R., “The Adaptive Object-Model Architectural Style”. URL: http://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.htm (2003)

Yoder, J. W., Balaguer, F., Johnson, R., “Intriguing technology from OOPSLA: Architecture and design of adaptive object-models”, December 2001 ACM SIGPLAN Notices, Volume 36 Issue 12

Listing 1: Runtime assembling of classes in Python 2.2+

```
class InvalidInputException(Exception):
    """The exception that's raised on invalid input"""
    pass

def generateClass():
    """Generates an empty class"""
    class Dummy(object):
        pass
    return Dummy

def generateDiscreteValidatingSetter(attr, allowedValues):
    """Generates a setter that validates discrete values"""
    def validateDiscrete(self, value):
       if value in allowedValues:
          setattr(self, '__'+attr, value)
       else:
          raise InvalidInputException
    return validateDiscrete

def generateRangeValidatingSetter(attr, lowerBound, upperBound):
    """Generates a setter that range validates a value"""
    def validateRange(self, value):
       if lowerBound <= value and value <= upperBound:
          setattr(self, '__'+attr, value)
       else:
          raise InvalidInputException
    return validateRange
```

```
def generateGetter(attr):
    """Generates a simple getter"""
    def getter(self):
        return getattr(self, '__'+attr)
    return getter

# Generate classes
BodyLengths = generateClass() # Composite class
LengthMeasurement = generateClass() # General length measurement

# Add validating properties to LengthMeasurement
LengthMeasurement.value = property(generateGetter('value'), \
    generateRangeValidatingSetter('value', 0, 300))
LengthMeasurement.unit = property(generateGetter('unit'), \
    generateDiscreteValidatingSetter('unit',['meters', 'inches',\
    'feet']))

# Instantiate and use the classes the natural way
lengths = BodyLengths()
lengths.arms = LengthMeasurement()
lengths.arms.value = 30
lengths.arms.unit = 'inches'

# And now the same, the dynamic way
# Obtain property names/values from somewhere
propname = 'fullbody'
valuepropname = 'value'
valuepropvalue = 1.85
unitpropname = 'unit'
unitpropvalue = 'meters'
# Use them
setattr(lengths, propname, LengthMeasurement())
setattr(getattr(lengths, propname), valuepropname,
valuepropvalue)
setattr(getattr(lengths, propname), unitpropname, unitpropvalue)

# Assign invalid values to properties
lengths.fullbody.value = 500 # Exception
lengths.fullbody.unit = 'yards' # Exception
```

Adaptive Object-Models

Peter Swart
Student Colloquium Group
Department of Computer Science
University of Groningen
p.swart@student.rug.nl

Nico van Benthem
Student Colloquium Group
Department of Computer Science
University of Groningen
n.van.benthem@student.rug.nl

Abstract

Current software tools let developers model a software system and generate program code which implements the modeled system. Using this method, there is a time delay between changing the model and executing its implementation, which degrades the flexibility and run-time adaptability of the software system. Adaptive Object-Models interpret the model of the system at run-time. The architecture of Adaptive Object-Models uses several methods to eliminate the time-delay and let changes to the model affect the system immediately.

The UML Virtual Machine is based on Adaptive Object-Models and provides an architecture for executing models. The use of UML as its input modeling language and its implementation imposes new concepts and limitations of the UML Virtual Machine in relation to Adaptive Object-Models.

Keywords

Adaptive Object-Model, Adaptive Systems, Meta-modeling, Meta-data, Causal Connection, UML virtual machines, logical architecture, physical architecture.

1 INTRODUCTION

Traditionally, developers model a software system using a modeling language like UML [4] or OPEN [2]. The resulting model is then used to generate program code, which implements the modeled system. When the model changes, the program code must be regenerated, the system must be reinstalled and reconfigured in order to make these changes effective. With the generation step being time consuming, rapid model prototyping is almost impossible. This time delay also makes optimization hard and often results in not fully optimized systems.

Nowadays most information systems need to be highly dynamic and flexible in order to meet their customers business needs. To be able to make changes to the system without having to write new code, certain aspects of the model can be disconnected from the code. For example, instead of implementing business rules in the code, they can be stored externally in a database or XML files.

Systems where the description of the model representing the changing needs of the user are interpreted by the system at runtime are called "Adaptive Object-Models" [3]. When the object model is changed, the systems behaviour is affected immediately.

A "UML virtual machine" [1] is based on Adaptive Object-Models and uses the UML language as its modeling language After discussing the concept of Adaptive Object-Models in more detail, this paper illustrates how to embed UML in the meta-level architecture of Adaptive Object-Models. It also addresses the new concepts and the resulting limitations of the UML virtual machine in relation to Adaptive Object-Models.

2 ADAPTIVE OBJECT MODELS

In the architecture of traditional object models, different business entities are usually modeled by different user classes. Code-generation is used to generate programming-level classes from their corresponding modeling-level classes. In order to make changes in business model affective, regeneration of the code is needed.

In Adaptive Object-Models, business entities are represented by instances rather than classes. This way, new entity types can be created at run-time by making new instances of a generic class. To support this, the architecture of Adaptive Object-Models has to deal with several issues:

- Subclasses model the small differences between classes and represent the changing business entities, so the number of subclasses is unknown.
- Although we cannot change a class without having to change the code, their number and type of attributes must be able to vary.
- Like attributes, business rules also have to be able to vary without having to change the code.
- In order to be able to change entity-relationships easily and immediately we need a way to separate associations from attributes.

After a short description of the classical framework for meta-modeling we will describe how to handle each of these issues in the following subsections.

2.1 Meta-modeling

The classical framework for meta-modeling is based on an architecture with four metal-ayers [6]:

- The information layer (M0) contains all the objects that are currently instantiated, also known as user objects.
- The model layer (M1) which contains the objects describing the objects in the information layer, also known as user classes.
- The meta-model layer (M2) contains the objects describing the used modeling language (e.g. UML).
- The meta-meta-model layer (M3) which defines the objects used to represent the modeling language.

In the traditional code-generation approach the modeling language (M2) is used to model the business entities (M1). Then tools are used to generate user classes (also M1) which can be instantiated at run-time (user objects, M0). This results in a disconnection in the meta-modeling framework between modeling-level classes (M1) and the user objects (M0, see figure 1). Instead of generating user classes from

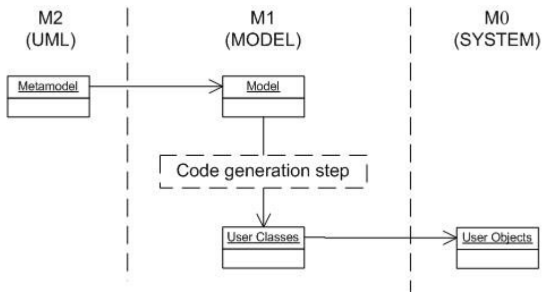


Figure 1: The disconnection in the code generation approach.

the descriptions of business entities, Adaptive Object-Models make instances of a generic class to model these business entities. New entities can now be created or changed at run-time, affecting the system immediately. This eliminates the disconnection between layers M1 and M0.

2.2 Subclasses

Object-oriented systems generally use the concept of subclassing in order to model the small differences between similar business entities. For example, a book store has to deal with different book types. All types have similar properties but also properties specific for their type.
In normal object models subclasses can be created as shown in figure 2. Each subclass inherits the attributes from its superclass Book and has its own, specific attributes.

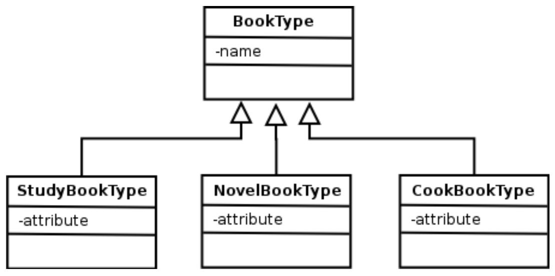


Figure 2: Traditional approach.

In Adaptive Object-Models, the *TypeObject*-pattern [3] is used to make subclasses simple instances of a generic class. This way, we can create an generic class `BookType` which defines all common properties for its subclasses. Each instance of `BookType` represents a new subclass having the same properties as its generic class (figure 3). To create a new subclass `NovelBook` at run-time, we can make a new instance of `BookType` and set the `name` property to novel.

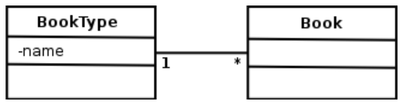


Figure 3: Using the *TypeObject* pattern

To instantiate a subclass we would need to create an instance from an instance. Instead, *TypeObject*-pattern requires another class `Book` to represent these instances. So to make an instance of the subclass `NovelBook` we have to create an instance of `Book` and link it to its corresponding subclass (see figure 3).

2.3 Attributes

Now we can create subclasses dynamically but the attributes of a subclass are still fixed. If we want to add an attribute, we would still have to change the code. The solution to this is to make a separate class `PropertyType` with attributes for the name and type of the attribute [3]. To add an attribute to a subclass we have to make a new instance of PropertyType and link it to its corresponding subclass.

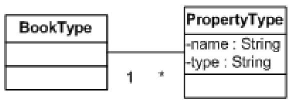


Figure 4: Property Types

For each new instance of the class Book we now also have to create an instance of Property to hold the values for each instance of PropertyType that is linked to its subclass.

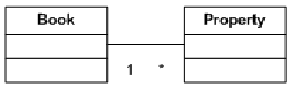


Figure 5: Properties

2.4 Rules and Algorithms

Policies and constraints on a system are called business rules. These business rules affect the behaviour of the system. They describe which values of attributes are valid and specifies when an algorithm can be or must be executed. Because we want to be able to create and change business rules at run-time we cannot implement them as methods of a certain class. Because we use a single class Book to create instances of different types of books, all these methods would have to be in this class.

When we use a Rule object to implement these algorithms, it is easy to change the behaviour by making or changing Rule objects at run-time [3]. For example, we can associate a Rule object to instances of the BookType class.
Complex business rules are combinations of primitive rules so two types of rules are possible. Primitive rules are elementary rules and composite rules can be specified by a set of

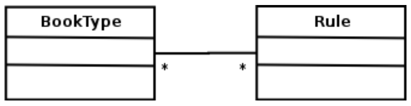


Figure 6: Adding rules.

primitive rules. This can be modeled by making subclasses of the Rule object.

2.5 Entity-Relationships

Relationships are properties of entities that associates it with another entity. Although there are several ways to make a distinction between properties and associations as attributes, in Adaptive Object-Model designs, associations are often represented as objects [3]. These objects can be created and changed during runtime in order to adapt to the changing business environment. Figure 7 illustrates how an association between a book and an author can be created using an instance of the Association object.

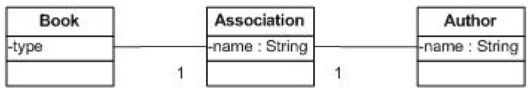


Figure 7: Association as an object.

Similar to attributes, we can link instances of a AssociationType to the corresponding instance of BookType to specify possible entity-relations between two subclasses.

An association object always represents a relation between two objects, but the cardinality between these object should also be specified. In our example, we would need extra attributes in the association object to specify that a book can only have one author.

3 UML VIRTUAL MACHINE

In the previous section we described the architectural style of Adaptive Object-Models. In this section we will describe the UML Virtual Machine [1].

It is based on Adaptive Object-Models and uses UML as its modeling language. We will discuss how its architecture relates to the architecture of Adaptive Object-Models and what new concepts it has. Besides the new concepts, we will also discuss the limitations of the UML Virtual Machine in relation to Adaptive Object Models.
The object of Adaptive-Model systems in general is to make a causal connection between meta-layer M1 and M0. A causal connection is achieved when changes in the upper layer immediately affect the underlying layer. The UML Virtual Machine takes this one step further: it realizes a causal connection between all four meta-layers [1].

3.1 Architecture

Riehle [1] divides the architecture of the virtual machine in two parts:

- The logical architecture describes how the causal connection between all four layers is obtained. The classes of all four layers are represented by instances. For example. UML objects like `Association` or `Generalisation` are represented as instances of `MetaClass` in the logical architecture.
- The physical architecture describes the physical classes that can be instantiated to represent the logical objects. The physical objects can be described in a object-oriented programming language like Java.

Figure 8 shows how the two layers relate to each other. In our example, the user class `Book` object is a physical instance of `Class` and the user object `book12345` is a physical instance of `Element`.

3.2 Implementation

This section describes implementation issues of Adaptive Object-Models and how the UML Virtual Machine deals with these issues.

3.2.1 Behaviour modeling

Although modeling languages like UML are able to describe object models, the modeling of a system's behaviour is less supported. Adaptive Object-Models use the concept of `Strategy` objects to add algorithmic detail through business rules . `Strategy` objects are used to define validation an operations on business entities at runtime.

To describe the complete behavior of a system based on the UML Virtual Machine, the UML state charts technique is used to describe the state and transitions of each entities in the system. In addition, the Object Constraint Language (OCL) is used to model constraints on elements, like business rules.

But even with these additions UML is not specified enough to be fully executable. Because UML is a modeling language we still need something to add algorithmic detail [1]. To add these details as hand-programmed classes, a well-defined extension architecture [8] is part of the virtual machine architecture. Because hand-programmed classes are needed, there is still a disconnection between meta-layers M1 and M0.

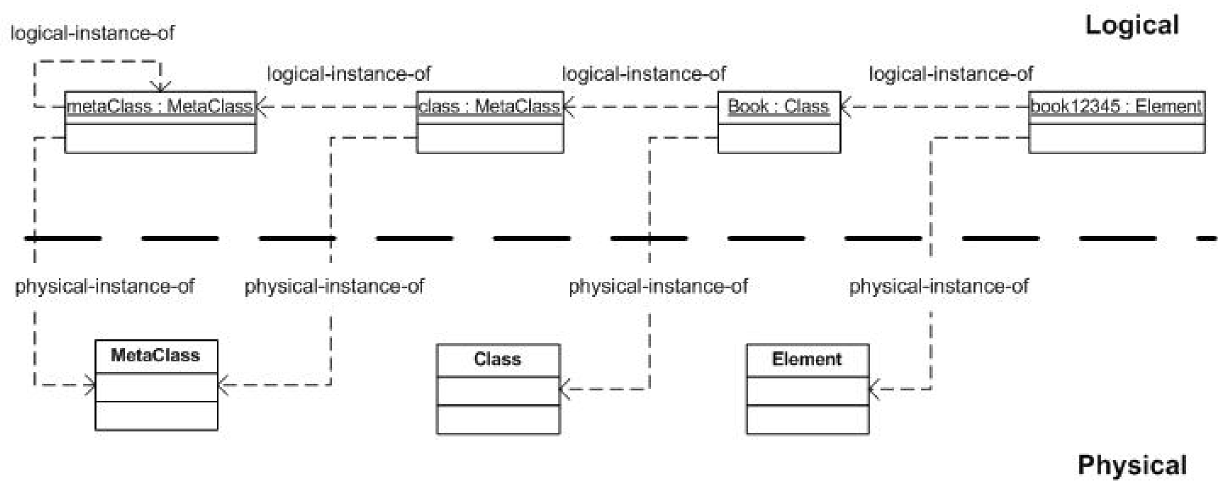


Figure 8: Relation between the logical and physical layer

3.2.2 Storing the model

In Adaptive Object-Model systems the user model isn't part of the code but is stored externally. Different kinds of databases can be used or the meta-data can be stored in an XML-file as long as the system can read and interpret it at run-time.

With the implementation of the UML Virtual Machine, XMI [5] is used to specify the model and its behaviour in UML. This does not add any extra limitations to the modeling capabilities of UML.

3.2.3 Understanding Adaptive Object-Models

With the design of Adaptive Object-Models, developers have to make a system that interpreters the model, rather than to implement it. This is probably unconventional for most developers, making the model difficult to understand and mistakes can easily be made. To support developers, editors and programming tools are made to assist them.

To assist business experts in changing the system model and behaviour at run-time, user interfaces must be created to specify new types or strategies.

UML is a common accepted modeling language and therefor suitable as input language for an Adaptive Object-Model system.

3.2.4 Performance

The concept of Adaptive Object-Models eliminates the time delay between changing the model and affecting the running system. However this time delay is eliminated by interpreting the model. At run-time, interpreting a model will be slower than generating and executing the code.

In the architecture of the UML Virtual Machine all logical object are instances of `Element`. Because it would be inefficient to re-specify shared constraints for each logical object, large parts of the UML classes are implemented as subclasses of `Element`. Furthermore UML classes like Attributes and Associations are replaced by more efficient key objects.

3.2.5 Run-time Environment

Because UML itself is not (yet) a programming language, it needs a dedicated run-time environment to execute models. This run-time environment will result in limitations of what can be executed. The UML Virtual Machine currently uses Java as its implementation language.

4 RELATED WORK

In section 2 we discussed the `TypeObject`. [7] highlights how to decouple instances from their classes so that those classes can be implemented as instances with an detailed example.

The UML virtual machine is an architecture for an Adaptive Object-Model bases on UML and aimes for Java as its implementation language. [1] gives a more detailed description of the architecture along with its implementation.

A site [9] of Joseph W. Yoder links to several example implementations of the Adaptive Object-Model using Smalltalk or Java.

5 CONCLUSIONS

An Adaptive Object-Model is a system that represents classes, attributes, and relationships as meta-data. The business model, along with its business rules and policies, isn't part of the code but is stored externally to be interpreted at run-time. The basic idea is that user classes can be created at run-time to specify or change business entities at run-time. In the architecture of the UML Virtual Machine all logical classes are instances of physical classes and can be created at run-time. Developers have to build a machine that executes a model rather than to implement the model itself.

Because UML isn't specified enough to describe the behaviour of the system, algorithmic details have to be added by hand-programmed classes. This disconnection between the model and its execution conflicts with the idea of Adaptive Object-Models.The dedicated run-time environment needed to execute UML also imposes limitations of what can be executed.

References

[1] Dirk Bucka-Lassen Dirk Riehle, Steve Fraleigh and Nosa Omorogbe. The architecture of a uml virtual machine. *OOPSLA '01*, pages 327–341, 2001.

[2] Ian Graham Donald G. Firesmith, Brian Henderson-Sellers and Meilir Page-Jones. Open modeling language reference manual, 1998.

[3] Ralph Johnson Joseph W. Yoder, Federico Balaguer. Architecture and design of adaptive object-models. *ACM SIGPLAN Notices archive*, 36:50–60, 2001.

[4] OMG. Omg unified modeling language specification 1.3, 2000. Available from www.omg.org.

[5] OMG. Omg xml metadata interchange (xmi) specification, 2000. Available from www.omg.org.

[6] OMG. Metaobject facility specification v1.4, 2002.

[7] Bobby Woolf Ralph Johnson. The type object pattern, 1997. Available from www-lifia.info.unlp.edu.ar/fer/classof.html.

[8] Dirk Riehle. Framework design: a role modeling approach, 2000.

[9] Joseph W. Yoder. Joe's metadata and adaptive object-model pages, 1998.

A Survey on Taxonomies of Software Evolution

Wouter Storteboom and Arjen Vellinga

Institute for Mathematics and Computing Science
University of Groningen
P.O. Box 800, 9700 AV Groningen, The Netherlands
<w.storteboom, a.f.vellinga>@student.rug.nl

Abstract
Software evolution was an area that was hardly explored. Nowadays the impact of software evolution is becoming known. In this article an overview will be given on the current state of the taxonomy of software evolution. With the taxonomy, a framework can be constructed for analyzing the software evolution, models and tools. With the taxonomy of Felici a good starting point for the analysis of the software evolution is given. Mens et al. proposes a practical taxonomy for the evaluation of software tools for tracking software changes. These taxonomies complement each other; Felici gives the analysis of the evolution and Mens et al. provides the tools to gather data.

Keywords: taxonomy, software evolution, framework, change, analysis, overview

1 Introduction

Today a world without computers and software systems is not imaginable. We depend on major systems like banking, flight control, and telecommunications. It is not reasonable to think that once those systems were designed, they would never change. The software systems operate in a fast evolving world where new technologies are developed, and requirements for a particular system change with time. It also becomes apparent that as computer based systems become more and more complex, that the impact of changes has to be analyzed in order to minimize the consequences. Fixing a bug late during development is known to cost more then one fixed early in development. Software configuration management deals with the procedures of making a change in the software system, but not with changes in the environment or the people working with the system. Also, there is little information on how effective the software configuration management was.

The change of software is called software evolution. Software evolution is regarded as inevitable and needed by [1], although information about the subject is scarce. Software evolution is almost ignored by software engineering literature. A survey in [2] showed that just one of a hundred software engineering books dedicated a separate chapter on software evolution.

Without a proper definition and classification of software evolution it is hard for people to talk about the subject, or to use it. So a taxonomy (division into groups with similar attributes) of software evolution can help in understanding it. With a taxonomy, a conceptual framework can be identified. Software evolution, evolution models, formalisms and concrete tools can be analyzed and compared when using such a framework. This framework could be used to evaluate tools for gathering software evolution data (i.e. a versioning control system).

The goal of this paper is to give an overview on different taxonomies of software evolution. The three proposed taxonomies by Verhoef, Felici, and Mens will be explained in section 2. In section 3 the proposals will be discussed and compared. Section 4 concludes this paper.

2 Three proposed taxonomies

In this section three proposed taxonomies on software evolution by Verhoef, Felici, and Mens will be described.

2.1 “Software evolution: a taxonomy” by Verhoef

In [3] a possible definition of software evolution is introduced: *Software evolution concerns any change that is being made to the entire set of programs, procedures, and related documentation associated with a computer system that makes up a software system*. According to Verhoef there are three major aspects for software evolution, namely: software maintenance, software enhancement, and software life-cycle enabling.

The next sections describe these three major aspects.

2.1.1 Software maintenance

In [3] software maintenance is divided into five types:

- Predeliver maintenance: The activities to support future maintenance and ensuring supportability (i.e. automatic update down loader).
- Corrective maintenance: Maintenance that is needed to correct actual errors (i.e. bug fixing).
- Adaptive maintenance: Maintenance of software due to the changing environment in which it operates (i.e. porting to another platform).
- Perfective maintenance: Optimization of the software, performance, and its documentation.
- Preventive maintenance: All the activities to prevent faulty behavior due to unknown circumstances (i.e. adding exception handling).

2.1.2 Software enhancement

Software enhancement can be categorized into five types according to [3].

- Block functions: Adding new features to an existing system without causing extensive internal changes.
- Modified blocks: Make internal changes to the software system to extend its functionality.
- Modification and deletion: Replacing an obsolete feature with a new one.
- Scatter updates: Multiple new features that cause extensive internal changes throughout the whole software system.
- Hybrid enhancement: Multiple enhancements of poorly structured legacy systems.

2.1.3 Life-cycle enabling

Life-cycle enabling is according to [3] associated with legacy software and software renovation. It consists of all the previous activities described in sections 2.1.1 and 2.1.2.

It should be noted that paper [3] is not yet complete.

2.2 "Taxonomy of Evolution and Dependability" by Felici

In [1] Felici describes a framework to aid in the analysis of software evolution. Felici regards evolution of computer-based systems as a two-fold concept. On one hand it is an inevitable and needed aspect of computer systems. On the other hand the degradation of the dependability of computer-based systems may be due to evolution.

Felici defines the evolutionary space as shown in figure 1.

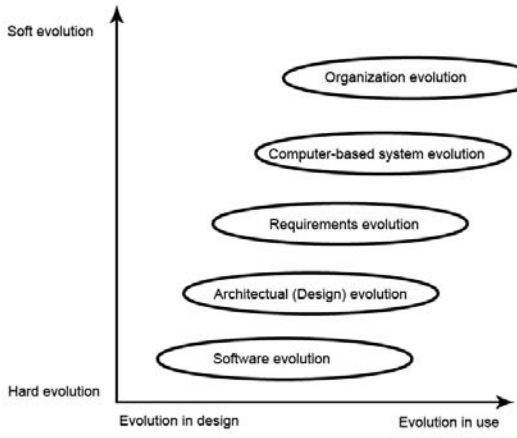


Figure 1 - Evolutionary space

Horizontally the life-cycle of a software system is displayed. This temporal dimension stands for the design, deployment, use of the computer system and eventually the demise. Vertically the physical dimension, where the evolution takes place, is stressed. Five evolutionary phenomena can be seen in figure 1 which are:

- Software evolution
- Architecture (Design) evolution
- Requirements evolution
- Computer-based system evolution
- Organization evolution

Although figure 1 shows the evolution phenomena individually, there might be overlap. So evolution could take place at more than one level at the same time. The different phenomena are described in the following subsections followed by a conceptual framework that is constructed by looking at the dependability relations.

2.2.1 Software evolution

Three patterns have been recognized by [1].

- Software tectonics emphasizes that software systems need to accommodate arising changes. This includes fixing bugs, but also fixing errors early in design. So software has to be implemented in order to support software evolution. Software systems should be adaptable to requirement changes by a series of small and controlled steps. By this, degradation of the software could be avoided.
- Flexible foundation says that the basics (tools, language, framework) where a software systems is made of, should be able to evolve also.
- Metamorphosis pattern: With this pattern, systems have mechanisms that allow them to manipulate their environment dynamically.

2.2.2 Architecture (design) evolution

When designing a software system the evolution of the architecture is unclear. Crucial is the ability to predict the evolution, so the architecture should implement the most suitable trade off between generality and specificity. Evolution can be divided in three groups, namely *architecture* evolution, *component* evolution or a combination of *both*.

2.2.3 Requirements evolution

This form of evolution was regarded as a management problem, but is receiving more interest within Software Engineering nowadays. What follows, are structures to support the creation of models concerning requirements evolution regarded by [1]:

- Type of requirements: This will identify the stable and possible changing requirements.
- Dependencies between requirements may be redefined across subsequent releases in order to minimize them.

- Type of changes, like adding, deleting and modifying requirements, define how changes alter the specification of the requirements.
- Requirements trace ability provides further information to analyze requirements evolution. A combination of the above structures and the formal requirements specification can be used to reason about requirements evolution.

2.2.4 Computer-based system evolution

Computer-based system evolution emphasizes the human aspects within socio-technical systems. Socio-technical systems are systems where the interaction between humans and technology is modeled. Figure 2 shows how the relations could be between social and technical systems.
A few models have arisen to capture data on socio-technical systems. One such model is social learning. Social learning explains how humans perceive machines in order to acquire computational artifacts and accomplish specific tasks. Another model is distributed cognition.

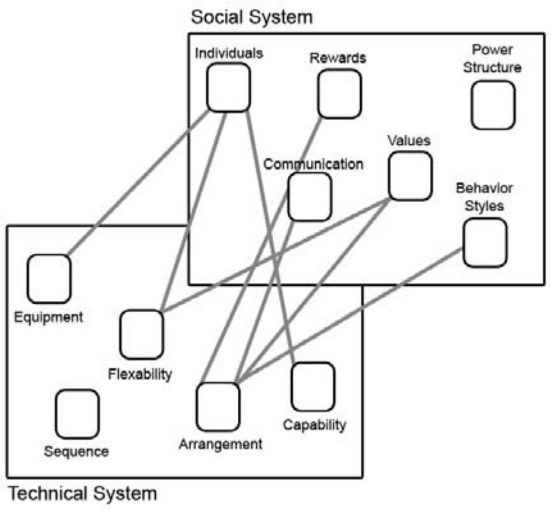


Figure 2

Distributed cognition recognizes the complex settings of socio-technical systems and analyzes how humans work, operate and create internal and external artifacts (e.g. rules, tools, representations, etcetera).

The above models are quite useful in order to look as a whole to analyze the evolution of computer-based systems. But they still need to be fully integrated with classical engineering methodologies, such as those described in the previous subsections.

2.2.5 Organization evolution

Due to the strong link between the social and technical evolution of computer-based systems they influence an evolution on organizational level. This aspect of evolution is not fully researched.

2.2.6 Dependability of evolution

The dependability of a software system changes as the software system evolves. Some hints that can be used to assessing the dependability are:

- Software evolution: Monitor software complexity; Identify the change-prone parts of the software; Carefully manage basic software structures; Monitor dependability metrics.
- Architecture (design) evolution: Assess the stability of the software architecture; Understand the relationships between the architecture and the business core; Analyze any (proposed or implemented) architecture change.
- Requirements evolution: Classify requirements according to their stability / volatility; Classify requirements changes; Monitor requirements evolution.
- Computer-based system evolution: Acquire a systematic view; Monitor the interactions between resources; Understand evolutionary dependencies; Monitor and analyze the (human) activities supported by the system.

- Organization evolution: Understand environmental constraints; Understand the business culture; Identify obstacles to changes.

The hints can be used to construct a conceptual framework for analyzing software evolution.

2.3 "Towards a Taxonomy of Software Evolution" by Mens et al.

In [4] a taxonomy of software evolution based on the characterizing mechanisms of change and the factors that influence the mechanism, is proposed. In this way the focus is shifted away from the purpose of the change (i.e., the why, and who question) and towards the underlying mechanisms. The taxonomy is organized into four logical groupings: *temporal properties*, *objects of change*, *system properties,* and *change support*. These groupings will be discussed further on.

The purpose of this taxonomy is manifold: (1) to position concrete software evolution tools and techniques within this domain; (2) to provide a framework for comparing and combining individual tools and techniques; (3) to evaluate the potential use of a software evolution tool or technique for a particular maintenance or change context and thus; (4) to provide an overview of the research domain of software evolution.

The proposed taxonomy focuses on the *when*, *where*, *what* and *how* aspects of software changes. These aspects follow from the logical groupings: temporal properties (when), objects of change (where), system properties (what) and change support (how) (figure 3).

The proposed taxonomy is not complete by any means. First of all, not all aspects of software changes are taken into consideration. The *who* and *why* aspects, for example, are not dealt with. Second, the proposed taxonomy is only one way of many to group software change mechanisms. And last, this taxonomy is

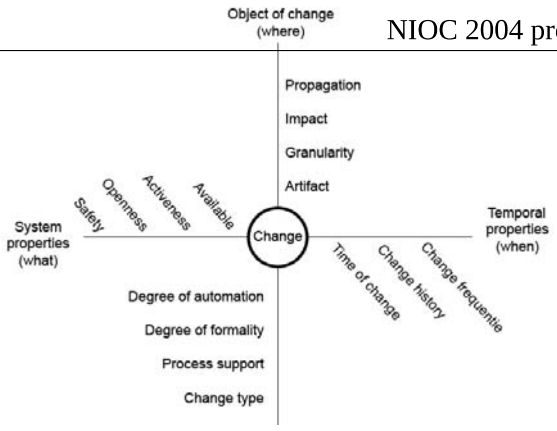


Figure 3

continuously evolving, since the basis elements are also evolving.

In the next subsection the four factors that define the evolutionary space, will be described.

2.3.1 Temporal properties (when)

The *when* question addresses the temporal properties that influences change support mechanisms. In [4] the following properties are mentioned:

- Time of change: The time of change in the software life-cycle will influence the kinds of change mechanisms that are needed. A software system that, for example, has to perform changes at runtime, must have a built in way to load a new component correctly.
- Change history: Change history has to do with all the changes that have been made to the software. This is what a version control system like CVS would do. If there was no version control system, changes would overwrite the previous ones and make it unable to keep track of the change history of the software.
- Change frequency: Changes to a software system may be performed *continuously*, *periodically* or at *arbitrary* intervals. If there are, for example, frequent changes, the change support mechanism has to be able to handle that. Otherwise it will become very difficult to roll-back the system to a previous version.

2.3.2 Object of change (where)

This grouping addresses the *where* question. Where in the software can changes be made, and which supporting mechanisms are needed for this? In [4] the following aspects of the *where* property are regarded:

- Artifact: Many kinds of software artifacts (man-made products) are subject to changes. This may vary from changes in the requirements, architecture, design, source code, documentation and test environments.
- Granularity: The granularity of a change defines the scale of the change. Coarse granularity could be changes to the entire software system or a subsystem. Medium granularity could be changes to classes. Fine granularity could be changes to local variables.
- Impact: The impact of a change is related to the granularity. Renaming a local variable is a local change, but renaming a global variable has global effects throughout the software system. The impact can also vary if there are changes in the level of abstraction that is used.
- Change propagation: A change in one part of the system can cause changes in other parts of the system. For this, mechanisms or tools are needed to help with analyzing how these changes propagate.

2.3.3 System properties (what)

The *what* question tries to answer what the software system has to go through in order to change. In [4] the following factors are mentioned related to the system properties:

- Availability: Most software system will continue to evolve during their lifetime. If systems have to be available the whole time, changes will have to be done at run-time. These systems will have to be designed differently from systems that do not have to be available the whole time. For them it is acceptable not to be available while changes are made.
- Activeness: Software systems can either be *reactive* or *proactive*. A proactive software system has internal monitors and logic to automatically change itself. A reactive system has changes applied to it externally.
- Openness: Software systems have a certain degree of openness. An open system is a system that was built with the idea of extensions. It usually has a framework that allows it to support extensions. An operating system is the perfect example of this. A closed system has no support for extensions. That does not mean it cannot be extended but changes are more difficult to make.
- Safety: A software system has to act flawless. Changes can cause unwanted behavior but a safe system would have safeguards for that. *Static* safety would ensure that there are no errors at compile-time. *Dynamic* safety is provided by a system to minimize errors and odd behavior at run-time by including code to prevent or restrict undesired behavior.

2.3.4 Change support (how)

During a change, various support mechanisms can be provided. These mechanisms help to analyze, manage, control, implement or measure software changes. The proposed mechanisms in [4] are:

- Degree of automation: Mechanisms to support software changes can be *fully automated*, *partially automated* or *completely manual*.
- Degree of formality: A change support mechanism can be based on a formalism or on an ad-hoc way of applying the change.
- Process support: Process support is the extent to which activities in the change process are supported by automated tools.

- Change type: *Structural* changes are adding, removing or modifying parts of the system. *Semantic* changes deal with the actual coding of the system. Other change types have been presented in [5]. The type of change influences the way the change is performed.

Future work will go into the usage and extension of the proposed taxonomy to be able to compare change support formalisms and processes.

3 Discussion

People are becoming more aware of the impact of software evolution and the need for a taxonomy of it. Verhoef, Felici, Mens et al. described different taxonomies. When comparing the three proposed taxonomies of software evolution in this paper it is clear that Felici [section 2.2] takes a more abstract approach. He defines the evolutionary space by just two dimensions, namely *when* and *where* the evolution takes place. In this space he places five evolutionary phenomena (software evolution, architecture evolution, requirements evolution, computer-based system evolution and organization evolution). With this rather limited classification the taxonomy stays more general in its description. On the other hand Mens et al. [section 2.3] takes a more concrete approach. More dimensions are used for defining the evolutionary space (the *when*, *where*, *what,* and *how* questions). By this more extensive arrangement of evolutionary phenomena a more detailed taxonomy is formed. That his taxonomy is more practical is proven by the evaluation in [4] of three tools by the proposed taxonomy.

Verhoef proposed in [section 2.1] a taxonomy of software evolution which is the least extensive of the three described. He only mentioned five forms of maintenance and five types of enhancements. In our view this is not a complete taxonomy, because maintenance and enhancements are also covered by the *change type* of the *how* question of Mens. Also other factors, like for instance temporal properties, are not evaluated by Verhoef.

In the previous section three taxonomies on software evolution were described, but only Verhoef proposed in [3] a definition: *Software evolution concerns any change that is being made to the entire set of programs, procedures, and related documentation associated with a computer system that makes up a software system.* This definition is not applicable to the other two taxonomies. Felici and Mens takes a lot more factors on software evolution into account. An example of this is that humans could also be regarded as a factor for software evolution. Software is developed for humans, so it is an important factor. The end-user of a software system learns when working with the software and humans also drive technical innovations which lead on their turn to software evolution. Mens et al. and Verhoef do not consider the human factor in their taxonomies.

Next to the differences between the taxonomies, there are similarities. The proposed taxonomy of Mens et al. describes a way to evaluate tools that support the gathering of evolutionary data. This is an implementation of the software that is needed to support software evolution, according to Felici [section 2.2.1]. The proposed taxonomy of Verhoef also adds to the software evolution of Felici [section 2.2.1]. Verhoef gives a more detailed description than Felici. The other parts of the taxonomy of Verhoef, could also add detail to the organisational evolution [section 2.2.5] when it is finished. But at the moment of writing this is not known. Overlap is present in these taxonomies and could make a contribution to a generic taxonomy of software evolution.

4 Conclusions

Software evolution was a neglected field of research but has grown. In order to have a common vocabulary for everyone to talk about software evolution, a taxonomy is the next step. Three proposed taxonomies are described and discussed.

In our opinion the proposal of Verhoef can not be regarded as a complete taxonomy. The maintenance and enhancement of software is covered by Mens et al.

The definition of software evolution given by Verhoef is rather limited; it is not applicable on the taxonomies by Felici and Mens. But we think that it is necessary to come with a better definition. With such a definition the evolutionary space can be defined better.

The taxonomy of Felici is an abstract taxonomy that makes for a good starting point to proceed with in research. The taxonomy of
Mens et al is oriented in a practical way. This taxonomy also gives an example of the actual use of software evolution analysis by evaluating three different tools. So at the moment the proposals of Felici and Mens et al complement each other and could be used together as one taxonomy. From this new taxonomy, further research can be done.

5 References

[1] M. Felici. Taxonomy of Evolution and Dependability, LFCS School of Informatics, The University of Edinburgh, United Kingdom

[2] C. Jones. Estimating Software Costs H10 p595-596, McCraw-Hill, 1998

[3] C. Verhoef. Software Evolution: A Taxonomy, University of Amsterdam, Amsterdam, The Netherlands

[4] T. Mens, J. Buckley, M. Zenger and A. Rashid. Towards a Taxonomy of Software Evolution, Vrije Universiteit Brussel Belgium.

[5] N. Chaplin, J. Hale, K. Khan, J. Ramil and W.-G Than. Types of software evolution and software maintenance, Journal of software maintenance and evolution p3-30, 2001

Efficient software engineering using software reuse

Bart Lemstra 1211196 and Roland Oldengarm 1211234

l.lemstra@student.rug.nl, r.oldengarm@student.rug.nl

Abstract
Software reuse is getting more important in software engineering. Systems are getting bigger and it would save a lot of time if existing software components (code, design, etc.) could be reused. In this paper a number of programming techniques are discussed which make software reuse easier. At the end a technique for component base reused is discussed.

Keywords: Software reuse, Separation of concerns, Aspect Oriented Programming, Subject Oriented Programming, Component adaptation techniques

1 Introduction

Software products are getting larger and more complicated. It is harder and harder to write good software programs totally from scratch. The primary goals of software engineering are to improve the quality of the software produced and to reduce the costs of construction and later on maintenance. Maintenance requires a comprehensible software product. A good way to achieve software comprehensibility and quality is to construct the software out of manageable pieces. This can be achieved by decomposing the software product into components. If the software is built up like this in a good manner, the different components can be reused in future software products. Also complexity is divided over the different components, so it is easier to adapt the product. Component based software engineering also allows existing components to be used in the product. In this paper we will discuss different ways how software can be built up from components and how these components can be reused in other software products.

2 Separation of concerns

Even in small software products contain lot of different units (units are e.g. classes). When performing some development task, the developer must be able to focus those units that are pertinent to that task and ignore all others. To accomplish this, software engineers identify *concerns* of importance, and seek to localize units representing concepts that pertain to each concern into a module. Ideally, one only need to look inside a module if one is interested in a given concern. For example, a class is a module containing units (describing methods and instance variables) that model a particular kind of object; all internal details of such objects, such as their representation, are described within the class. Separation of concerns of a software product will reduce the complexity of the software product. Also (parts of) the product can be reused and maintenance is easier to perform. Separation of concerns in multiple dimensions is discussed in [8]. They describe a new way how software artifacts can be modelled and implemented. This model allows separation of overlapping concerns along multiple dimensions. All current software formalisms support separation of concerns, using decomposition and composition. However, they provide only a limited set of decomposition and composition mechanisms. Separation of concerns in multiple dimensions claims not to have these limitations.
Software products consist of artifacts, such as requirements, design and code. Each artifact is being made up of units. E.g. in object-oriented programming each unit is a class. Units which contains smaller units are called compound units; the smaller units (which do not contain smaller units)

are called primitive units. The purpose of separating the product into different modules is *Separation of concerns*[7]. Even software products of moderate size are too large to be contained in one unit. That is why it is separated into different units. Ideally, when solving a problem, a software engineer can focus only on the relevant unit and can ignore all other units. Many kinds of concerns are important during the development, we call them dimensions of concern. Most common are data or object (data abstraction) or functional concerns (leading to separation into functions). Some concerns come from the domain in which the product is placed, others come from the requirements. E.g. object-oriented programming allows for decomposition to only a single dimension of concern, namely the data dimension. This is called the dominant dimension. In the formalism used, this dominant dimension should be stated.
However, decomposition along only one dominant dimension is in most cases inadequate. Modules and units become tangled and concerns are not separated well anymore. So, separation of concerns is not possible anymore. In [8] hyperslices are introduced to solve this problem.

2.1 Example: SEE

Because hyperslices are not easily understood, we give a short example. We take a look at the construction of a simple software engineering environment (SEE) for programs consisting of expressions. We assume a simplified software development process, consisting of informal requirements specification in natural language, design in UML, and implementation in Java. In short, the requirements are: *The SEE supports the specification of expression programs. It contains a set of tools that share a common representation of expressions. The initial tool set should include: an evaluation capability, which determines the result of evaluating an expression; a display capability, which depicts an expression textually; and a check capability, which checks an expression for syntactic and semantic correctness.*

2.1.1 Evaluating the SEE

In [8] the system is now evaluated by looking at the system and how it reacts on possible future changes. We will sum up their general conclusions:

Impact of change The goal of low impact of change requires *additive* (adding components or code, e.g.), rather than *invasive* (changing existing code). Simple changes often have a widespread and invasive when applying them to for example the expression SEE.

Reuse Many people see reuse as the holy grale, because it should be the answer to the growing complexity of programs. But reuse is currently often limited and only used on code, not on requirements or designs. Part of the impediment to large-scale reuse is that larger artifacts entail more design and implementation decisions, which can result in tangling of concerns and coupling of features, thus reducing reusability.

Traceability Different artifacts are written for different purposes and include different levels of abstraction. Thus, they are specified in different formalisms and are often decomposed and structured differently. Developers must create connections among related artifacts explicitly (e.g. in [3]). These connections are complex and can be invalidated readily, and more important, they do not solve the problem of tangling or scattering.

The other conclusion is that the main reason for these problems is the *tyranny of the dominant decomposition*, as said before. Using current programming techniques, only support a small set of decompositions and usually only a single dominant one at a time. This dominant decomposition satisfies some important needs, but usually at the expense of others. For e.g., when decomposing to minimalize the impact of future changes, the traceability may decrease.

2.1.2 Breaking the tyranny

To achieve the full potential of separation of concerns, we need to break the tyranny of the dominant decomposition. If a system could be modularized according to all possible concerns (e.g. features), the problems described would be solved. Hyperslices are a solution; we will use the example of SEE to explain hyperslices.

2.2 Hyperslices

Hyperslices are a set of conventional modules, each module is written in any formalism. Hyperslices are intended to encapsulate one dimension of concern other than the dominant dimension. The modules and units that are contained thus only pertain a given concern. It can occur that one unit is in more than one hyperslice, thus hyperslices can overlap. A system is built up from multiple hyperslices. There will be a hyperslice for each necessary dimension of concern. It is not required new artifact formalisms have to be used, so the engineer can use their familiar formalisms. The modules inside a hyperslice are standard modules, except that they only contain the units pertinent to the concern of the hyperslice. This can give problems in e.g. Java, which require that all methods are declared. However, because hyperslices are composed later on, the missing methods will be combined again. The system is built up from hyperslices, thereby separating all the concerns of importance in the system.
To make it more clear we take a look at how SEE, the example from the previous section, will look like in terms of hyperslices:

2.2.1 Composing hyperslices

When we have defined the hyperslices, they have to be composed. This is achieved by hypermodules, a hypermodule is a set of hyperslices. Each hypermodule has a composition rule that specifies how the hyperslices should be composed. The composition rules have to be defined by the engineer. Note that the complete system is also a hypermodule, consisting of all artifacts.
Composition is based on commonality of concepts across units. Units with the same concept are combined. This is done in three steps:
Matching the concepts of the units in different hyperslices, *reconciliation* of differences in these descriptions and *integration* of the units to provide a unified whole. The rule stated in [8] comes from their research into subject oriented programming [2; 6]. Their approach is one general rule and specific rules for exceptions on the general rule.
An alternative is to define different rules for each hyperslice, thus let each hyperslice specify how it is composed. If it is possible that hyperslices can refer to other hyperslices, coupling is increased and thus reusability will decrease. Other hyperslices can be referred to come to a composition. If not, flexibility is decreased, but reusability is increased. The solution is to put the rule a level higher, in the hypermodule. This manner allows for enhanced flexibility and flexible overlap.

2.3 Summary

Summarizing hyperslices, each artifact is written as a hypermodule. For each concern of importance that can not be encapsulated using artifacts effectively using the artifact formalism, a hyperslice is introduced. Composition rules are written which define how the hyperslices are combined together. An enclosing hypermodule is defined which encapsulates the entire system.

3 Aspect-Oriented Programming

Using object-oriented programming (OOP) it is very difficult to capture all the important design decisions a program must implement. In [4] a new programming technique, aspect-oriented programming (AOP) is presented. The issues that design decisions address are called aspects. AOP makes it possible to clearly express programs involving such aspects.

In general, whenever two properties being programmed must compose differently and yet be coordinated, we say that they cross-cut each other. Because existing programming languages provide only one composition mechanism, the programmer must do the co-composition manually, leading to complexity and tangling in the code. An aspect is a property that has to be implemented and can not be cleanly encapsulated in a generalized procedure. A component on the other hand, can be encapsulated in a generalized procedure. The goal of AOP is: to support the programmer in cleanly separating components and aspects from each other. Existing programming languages only allow the programmer to separate components, and not aspects. When using an existing programming language, these aspects become intangled in the code. This could be solved by using AOP.

Aspect-oriented programming allows for appropriate isolation, composition and reuse of the aspect code. It is a new technique, and only at the beginning stage. A lot of things still have to be researched, but AOP is a very promising idea.

4 Subject oriented programming

In classical object-oriented programming the different views of the users of the system are not taken into account. In [2] they introduce a new object-oriented approach, called subject oriented programming. When we look at a tree, the class in classical object-oriented programming would have for example the following properties:

Properties Height, weight, density

Actions Grow, photosynthesis.

But e.g. a tax-assessor has his own view which differs from the class above. A property he may be interested in is assessed value when cutting the tree. To solve this problem, the designer of the tax-assessor application can do it two ways:

- Using encapsulation and polymorphism which inherits all methods and properties of the superclass `Tree` but implements new methods especially for the tax-assessor.
- Integrating the methods and properties which are particular for the tax-assessor into the class `Tree`.

The latter is unmanageable, because the tax-assessor was just an example. Other applications (application in terms of a program), such as a bird or a gardener, have their own methods and properties too and have to be integrated in the class `Tree`. This will result in an unmanageable large class `Tree`.
Note that the names tax-assessor, bird and gardener are just random chosen. The tree also could have been a node in a parse tree and the bird the compiler.
The first solution is better than the latter, but there is one big disadvantage. To use the advantages of polymorphism and encapsulation, the designer has to cope with a ever-expanding collection of methods and properties. Because not all programming is done in-house, the designer has to implement all future requirements at forehand. This is, as you can imagine, impossible. Subject-oriented programming may be the solution.

4.1 Overview

The overall goal of subject-oriented programming is to facilitate the development and evolution of suites of cooperating applications. Applications share objects and jointly contribute to the execution of the program. The following requirements should be met:

- It must be possible to develop applications separately and compose them later on.
- The different applications should not be dependent on other applications they are to be composed with.
- The composed applications might cooperate loosely or closely, and might be tightly bound for frequent , fast interaction or be widely distributed.
- It must be possible to introduce new applications into the composition, without changing the other applications.
- Unanticipated, including new applications that extend the current applications in unanticipated ways, must be supported. This is discussed in more detail in [5].
- Within each application, polymorphism, encapsulation and inheritance should be maintained.

The link with object-oriented programming is discussed in more detail in [1].
In subject oriented programming each application is a collection of one or more subjects. A subject is defined as a collection of states and behaviors; thus a perception of the world as seen by a particular application or tool. Subjects are not the same as classes; they describe the behavior and states of many classes.
In classical object-oriented programming particular state and behavior are often thought of as intrinsic to an object. In subject oriented programming the developer is free to have a subject deal with this. Thus, the subject deals with the intrinsic properties of more than one object. Any manipulation of

an intrinsic property of a particular object should be handled by that object.
One essential characteristic of subject-oriented programming is that different subjects can separately define and operate upon different subjects, without any subject needing to know the details associated with those objects by other objects. Only object identity is necessarily needed.

A *subject activation* is nothing more than an executing instance of a subject, including the data which is manipulated. Subjects can also be composed. Composition rules has to be defined, they tell how the composition of the different subjects takes place.
An object identifier (or an *OID* in short) gives us a unique identification of the objects, global or in the context of a subject.

In classical object-oriented programming, an object model is the model seen by any subject. Within a subject, an object ha an implementation class, according to the needs of the subject. Thus, subject-oriented programming includes the classical object-oriented programming.
Interfaces describe which operations a class of objects supports. Because the underlying code is not known (it is seen as a black box), this leads to enhanced flexibility and software reuse. In classical object-oriented programming this is already often used. In subject-oriented programming this is essential, because subjects do not know the implementation of other subjects, only their interface.

4.2 Conclusion

Classical object oriented programming gives in some cases serious problems. Subject oriented programming, which uses classical object oriented programming can be the solution. A subject is a part of the whole system and the subject compositor puts these parts together to a whole.
Subjects consist of a number of classes and interact with other subjects through their interface. The subjects are seen as black boxes, thus their implementation is invisible to the outer world. This leads to greater flexibility and software reuse.

5 Component adaptation techniques

The aim of component reuse is to create a collection of reusable components that can be used for component-based application development. This can allow for faster and cheaper development. Existing reuse techniques however are rather naive. They assume that existing components can just be plugged into an application. This is of course not true. Often a component needs to be adapted in order to fit into a specific application. There are several traditional techniques for adapting a component.

5.1 Requirements

There are a couple of requirements that the adaption techniques should meet in order to be successful.

Transparent The adaptation should be transparent. This means that both the user of the adapted component and the component itself are unaware of the adaptation in between them.

Black-box The software engineer should be able to view a component as a black box. This means that the adaptation technique needs no knowledge of the internal structure of the component.

Composable The adaptation technique should be easily composable with the component for which it is applied. Also, the adaptation should be composable with other adaptations, it may be that a component needs multiple adaptations.

Configurable Adaptation techniques have to be sufficiently configurable in order to be useful.

Reusable The adaptation technique should also be reusable, so that the adaptation type can be reused in the future.

5.2 Different techniques

There are three convential component adaptation techniques, copy-paste, inheritance and wrapping. Using copy-paste, a software engineer just copy pasts parts of existing components. The engineer will often make changes to the code before actually using it. Copy-paste is a transparent technique. The requirements for black-box, composable, configurable and Reusable are not met. A second technique for white-box adaptation and reuse is provided by inheritance. Inheritance provides the important advantage that the code remains to exist in one location. However, one of the main disadvantages of inheritance is that the software engineer generally must have detailed understanding of the internal functionality of a superclass. This is also a transparent technique. Whether it meets the black-box requirement, depends on its implementation in the language model. The composability requirement is partly met. Configurability and reusability are hardly supported. Wrapping declares one or more components as part of an encapsulating component, i.e. the wrapper, but this component only has functionality for forwarding, with minor changes, requests from clients to the wrapped components. An important disadvantage of wrapping is that it may result in considerable implementation overhead since the complete interface of the wrapped component needs to be handled by the wrapper. Wrapping is not transparent and configurable, but is does support black-box and composability. The wrapper can be reused in those cases where exactly the same adaptation behavior is required.

Concluding, none of the conventional component adaptation techniques fulfils the requirements that are required for effective component-based software engineering.

5.3 Super imposition

Super imposition as a concept is a very suitable technique for adapting components in a component-based system. A component and the functionality adapting the component are two separate entities, but need to be very tightly integrated. Often, both the component and the adaptation technique are reusable entities. The combination of component and adaptation however, is in most cases too specific for reuse. So, in super imposition, in addition to a set of reusable components, there is a set of reusable component adaptation types. The adaptations should be configurable and composable. This should allow for complex component adaptations. Components can be adapted by more than one adaptation type.

A object o is defined as $o = (I, M, S, P)$ where I indicates the interface of the object, M the set of methods, S the state space formed by the instance variables and P the mapping from the interface to the methods. An object can have a set of superimposing entities g_n. An entity g_n can be composed with an object o. This results in another object o'. o' is an adaptation of o.

There are different component adaptation types: component interface changes, component composition and component monitoring.

5.3.1 Component interface changes

It is often the case that the interface of a component that is to be reused does not match the expected interface. Typical examples are that operations have the wrong names or that the interface contains irrelevant operations. In this case, the interface of the component has to be adapted. There are some typical examples of component interface changes:

Changing operation names Some of the names of the operations provided do not match the expected interface

Restricting parts of the interface A component may require the exclusion of a part of the interface. This part may not be relevant.

Client and state-based restriction A component may need to act in several roles. This requires the component to present a tailored interface to each client type.

5.3.2 Component composition

During the design of a system, an overview of the needed components is defined. When searching for suitable components, it may occur that there isn't a component that exactly matches the required component. It can however be the case that the required component can be composed of two or more components. There are three types of component adaptation relevant for component composition:

Delegation of requests If a component is not able to provide a required service, the component can delegate a request for such a service to another component that is able to provide the requested service.

Component composition Two components can be aggregated in a encapsulating component.

Acquaintance selection and binding Virtually all components require other components, acquaintances, to provide them with services in order to be able to deliver the functionality needed by the system.

5.3.3 Component monitoring

This category is, as the name implies, primarily concerned with the monitoring of the component so that other components are notified or invoked when certain events at the monitored component occur. There are again three examples of monitoring that can be superimposed on reusable components:

Implicit invocation The concept of implicit invocation is concerned with notifying relevant components, either directly by message sending or indirectly through event generation.

Observer notification This explains how the relation between some object and a set of objects depending on the state of that object should be implemented.

State monitoring In some cases, dependent components do not want to be notified for every state change in the observed component, but only when the component state exceeds certain boundaries.

5.4 Evaluation of super imposition

Super imposition meets all the requirements for an adaptation technique. It is fully transparent and fully black-box. Adaptation types can be freely composed with each other. The adaptation types are configurable and reusable. So super imposition is superior to the conventional adaptation techniques.

6 Conclusion

The main problem of software engineering in the future will be the larger and more complex systems that have to be built. To solve this problem, we have presented a couple of techniques in this paper that help in building better understandable and maintainable software systems. Subject and aspect oriented programming aid in designing and programming in a way that is much better for maintainability and reusability. Super imposition is a very promising technique for reuse. The ideas behind these techniques are good, but it remains to be seen if they will really work in practice. Only time will tell.

References

[1] W. Harrison, H. Ossher, and M. Kavianpour. Integrating coarse-grained and fine-grained tool integration. *Proceedings of Fifth Internation Workshop on Computer Aided Software Engineering*, 1992.

[2] W. H. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA*, pages 411–428, 1993.

[3] R. Kadia. Issues encountered in building a flexible software development environment: Lessons from the arcadia project. *Proceedings of the Fifth ACM SIG-SOFT Symposium on Software Development Environments (SDE5)*, pages 169–180, 1992.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[5] H. Ossher and W. H. Harrison. Combination of inheritance hierarchies. In *OOPSLA*, pages 25–40, 1992.

[6] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifiying subject-oriented programming. *TAPOS*, 1996.

[7] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[8] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

An introduction to Software aging: How to deal with it?

Rodney Heinkens, Onno de Graaf
R.Heinkens@Student.RuG.nl, O.de.Graaf@Student.RuG.nl
Dept. of Mathematics and Computing Science, University of Groningen

Abstract. *Even though software is a mathematical product, it can still age. In this paper, we identify two types of software aging, namely aging due to updating a software product and aging due to the lacking of updating the software product. We discuss several causes and corresponding causes related to software aging. Finally, we will propose several methods to deal with software aging. Our conclusion, however, is that software aging is inevitable. The proposed methods only delay the aging process.*

Keywords. *Software aging, software architecture, software design, design decision, rationale, architectural erosion, architectural drift, customer requirements/desires.*

1 Introduction

In this article, we will discuss the notion of software aging. As software products grow older, new technologies arise. Such technologies could offer new functionality, which could lead to totally different designs for software products. Object-oriented programming languages are examples of such innovative technologies. Besides these new technologies, the desires from customers also change. For instance, customers could ask for more functionality or better performance. If your software product is not capable of sufficing these desires, often there are other software products which do suffice their desires. So there is a need to update one's software product. However, by adapting your software product according to the desires of the customer, architectural erosion could take place. Architectural erosion, sometimes referred to as design erosion, is what we define as a decrease in the structure and quality attributes of the software product. Often, the impact on short-term is little, but for long-term, the impact could be dramatic. Architectural erosion is one aspect of software aging. If you do not update your product according to the desires of customers, one has to cope with software aging of a different form, namely outdated software.

In this article we will discuss the long-term impacts of software aging. In section 2, we will discuss software architectures in more detail. We will use the concepts of software architectures to make architectural erosion more clear. In section 3, we will introduce different aspects regarding software aging. Sections 4 and 5 provide more insight in the causes of and problems that arise with software aging. In section 6 we will discuss some methods how the impact of software aging can be minimized. Finally, section 7 provides an overview of our findings in this article.

2 Software architectures

There is no standard universally-accepted definition of the concept software architecture. In [IEEE Standard P1471], the following definition of software architecture is given: "*Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution*". Based on this definition, we can identify some aspects of software architecture, which we shall discuss briefly in this section.

2.1 Design process

People designing software architectures are referred to as software architects. Software architects have to cope with all kinds of limiting factors. Noteworthy factors are:

- Time,
- Experience,
- Stakeholders,
- Available technologies & resources.

We will address these factors briefly. The first factor, time, is limiting the architect in taking all possible solutions into account. An instance of this factor is time-to-market, the time between the start of development and the actual launch of a product. If a software product is launched too late, it is likely that another vendor has gained a proportional share of the market. The choices the architect makes during the design process are referred to as design decisions, which we shall address in the following paragraph. Considerations that are made to come up with design decisions are referred to as the rationale, which we will address in paragraph 2.3. Since there is a need to achieve a short time-to-market, often, suboptimal design decisions are taken.

The second factor, experience, is influencing the decision making of the architect. If the architect has made a fault in the past, he probably will not make it again in the future. However, if one lacks this experience, it is prone to happen that wrong decisions are taken.

The third factor, stakeholders, are persons who have some interest in the software product at hand and have some power or influence [7]. Examples of stakeholders are end users and management. It happens more than once that different stakeholders have different requirements. For instance, the end users want good performance and ease-of-use, but the management does not want to make a too large investment in the software product. The negative consequences which result from conflicting requirements can also be enforced by internal company politics.

The final factor that will be discussed here, the available technologies and resources, can limit the architect in his considerations. An example of this is the use of the Internet, which would have been no option for an architect twenty years ago. Nowadays, it is a very common practice to make use of the Internet, for instance with network gaming. Examples of resources are intelligent development tools, like editors and build tools.

2.2 Design decisions

As was mentioned before, the design decisions are the decisions that are taken during the design of the architecture. During an architectural design, a series of design decisions are taken. Making one decision can exclude other decisions. An example of such a dependency is when choosing to develop for Windows, one cannot make use of a Linux API. The succession of design decisions can be regarded as a chain. If one design decision is changed, this could affect following design decisions drastically.

As with software architectures, there is no standard universally-accepted definition of design decisions. Also, for documenting these decisions there is no standard. This is one of the issues which we will address later on in this article.

2.3 Rationale

When making design decisions, one takes different options into account. The reason for choosing one option and declining the other options is referred to as the rationale of the design decision at hand. It can be referred to as a textual explanation of the "why" of a design decision [6] and it should be included in the design documents. The lack of documenting the rationale is one of the key issues when looking at software aging. This too will be discussed later on in this article.

2.4 Design documents

The design of a software architecture is often documented. For this purpose, one could use UML diagrams. These documents amongst others help for communication with the different stakeholders. For different types of stakeholders different types of design documents are made. These different types of design documents are a representation of the architecture and they provide a different view on the architecture [8]. This is necessary in order to make the architecture understandable for the different stakeholders who often want to know different aspects about the architecture.

3 Software aging

In this section, we will discuss the aging of software in general. We will use examples to illustrate it. In the succeeding two sections, we will discuss the causes and problems regarding software aging more thoroughly.

As mentioned earlier, software is designed by software architects. At a certain point in time, the software architect designing the architecture will have to make a design decision. Such decisions have a certain impact on the quality attributes of the software product at hand. Examples of such attributes are performance and reliability. When one desires high performance, this implies that one needs to pay with some other quality attribute [9], for instance maintainability. After the software product is being used for a while, the customers might want to have some functionality added to it. To implement these changes, the original design will have to be modified. It is not unlikely that the redesign will be performed by a different software architect than the one who designed the original software product. This is where the problems start.

The original architect took multiple design decisions when designing the architecture of the software product. These design decisions affected the quality attributes of the software product. However, it can be very hard for the new architect to understand all the design decisions that were taken during the earlier design. Especially the rationale behind such a decision needs to be understood well, which unfortunately often is not the case. To even further complicate the matter, changing design decisions could affect other design decisions later on in the chain.

However, by not understanding the decisions and corresponding rationales, it is very likely that wrong decisions are taken, based upon a miscommunication between the new architect and the documentation provided by the earlier architects, if any. The rationales behind design decisions provide insight to why such a decision was taken. Too often, these design decisions and rationales are not provided in documents. Results of mismatches in the rationale of the original architect and the rationale of the new architects result in reductions of the overall quality of the software attributes. This is an instance of architectural erosion.

In [4], two definitions are given, which we will use in this article also. First, they define architectural erosion as follows: "violations in the architecture that lead to increased system problems and brittleness". Second, architectural drift is defined as "a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture".

In the following two sections, we will make use of the aforementioned concepts, in order to illustrate the causes and problems of software aging.

4 Causes of software aging

In this section, we will discuss the causes of software aging. First, we will discuss two distinct types of software aging, as proposed by [2]. The first type is related to modifiability, or rather the lack of modifiability. Modifying software products is needed in order to meet the ever-changing requirements of the customers. The other type is related to the impacts that the performed changes have on software products. The combination of these two can lead to a value decline of the software product at hand. We now will discuss the causes of software aging, without trying to cover every aspect of it.

4.1 Modifiability causes

The first cause of software aging is caused by the lack of updating the software product with respect to its features. As was explained in the example in the introduction, one always has to improve the software to keep up with the desires of customers. If software is not updated frequently in order to make it adhere to customer desires, the customers will become dissatisfied. If this is the case, they will switch to a new software product, as soon as the benefits of using the other software product outweigh the costs of converting to that product [2].

An associated problem is the lack of time and how the architects have to cope with this. Since the software product needs to be kept up-to-date, changes have to be carried through as fast as possible. As a result of this, the architect will not have time for all of the necessary tasks for successfully keeping the software 'young'. Such tasks are for instance: taking time to understand the rationales of the design decisions, document the rationale of the new decisions, update the documentation, etc.

4.2 Architectural erosion

The software architect, who designed the original architecture, took the original design decisions. Often, the rationales why the decisions were taken as they were, are not documented (well enough). So, the next software architect who designs the modified software architecture, which is in turn based on the original architecture, does not fully understand the rationale behind the original design decisions. However, the software architect will have to make the proper design decisions based on his experience, his understanding of the software architecture and the documentation at hand. Mismatches in the rationales here could lead to architectural erosion and architectural drift.

The big problem with mismatches in the rationale is that it could easily lead to a vicious circle. Since the original architect's rationale and the second architect's rationale did not match up, the new architecture and the corresponding documentation, if any, will be even harder to understand. This in turn leads to more mismatches in the following update of the software product. Also, the structure becomes more complex. Hence, the structure is also harder to understand after every update.

An important cause which needs to be considered, is formed by the relations or dependencies between design decisions. As mentioned earlier, if one decision was taken, it could imply that some other decision, which appears later on in the decision chain, is no longer an option. This is for instance the case when one is developing for Windows and makes use of the Windows API, but wants to switch to Linux. The design decisions, which were made possible by the available Windows API, will have to be revised.

As a final cause of software aging, we mention how designers and engineers regard their own work. Sometimes it is the case that they are reluctant to criticism, since they cannot cope with the thought that their own work is not perfect. How could it be wrong? After all, they have designed it themselves. Some even consider their designs as works of art. However, not accepting your own mistakes will result in repetitive error making. This will be addressed more thoroughly in section 6.4.

5 Software aging problems

In this section, we discuss the problems that arise with the aging of software. We will use the same distinction that was used in the previous section, namely modifiability problems and architectural erosion.

5.1 Modifiability problems

The first type of problems has to do with the modification of the software product. As mentioned earlier, the updating of software products is necessary to keep up with the ever-changing requirements and desires of the customers. The two problems we will discuss here are the traceability of the design decisions and the increasing maintenance costs.

First, we will discuss the traceability of the design decisions. Since these decisions and the corresponding rationale often are not documented (well), one needs to trace these design decisions. As software products grow, and the size of the source code grows as well, the design becomes more complex also. As a consequence, tracing the design decisions becomes harder. However, when the aspects were not understood completely at first hand, the updated document will contain even less useful documentation. The harder these design decisions are to be traced, the more difficult it is to modify the software product properly.

This brings us to the second problem regarding modifying software products. Each time it becomes harder to understand the software, whether it is caused by poor documentation or more complex source code and corresponding designs, the maintenance cost rise. Since it is harder to maintain the product, designers and engineers will tend to use suboptimal solutions rather than the more complex optimal solutions. These suboptimal solutions only are less expensive on the short term, not on the long term. When a couple of suboptimal solutions are used, they can get in each others way. This is what some people refer to as smelly or rotting code [5], and what we see as an instance of architectural erosion.

5.2 Architectural erosion

Next, we will discuss the problems that arise regarding the software architecture. The two problems we discuss here are the reduced performance and the decreasing reliability. Both problems are related to architectural erosion.

First of all, we will discuss the reduced performance. Since the requirements and desires of customers change, the available features will have to adapt in order to comply with their requirements and desires. However, also the factor time plays an important role. This is for instance the case with the aspect time-to-market. If a product takes too long for development, another vendor will probably have a similar product launched earlier on. So, architects would most likely take the easiest route to implement the necessary features. Often, the easiest method to add features is to add new code to the current source code. However, then the source code grows larger by every added feature. The result of growing amounts of source code often is a dropdown in performance. However, not only the performance suffers here. Also, (more complex) growing source code results in less understandable code and hence, less optimal solutions will be taken for the following updates.

Another quality attribute that suffers from software aging is the reliability. When software is maintained, for instance adding features, not only new features are introduced, but also new errors in the source code are introduced [2]. If the product is not tested sufficiently, this results in a less reliable product. Keep in mind that testing in itself is not enough to ensure quality. The need for extensive testing results in higher maintenance costs as proper testing often takes a lot of time, and time is money. These additional costs bring us back to the earlier referred to problem, namely the rising maintenance costs.

Since multiple quality attributes suffer from the above problems, e.g. performance and maintainability, one could speak of architectural drift.

6 How to deal with software aging?

In this section we will discuss several methods for dealing with software aging.

6.1 Minimal vs. optimal design strategy

In an experiment [1], a program was created using both an optimal design strategy and a minimal design strategy. In that experiment, it became clear that a minimal design strategy speeds up the development, at least before the architectural design had eroded too much. However, even in the optimal design strategy, they experienced architectural erosion. Some design decisions have enormous impact on the code. Decisions to change other decisions which have such a great impact can be really troublesome. Therefore you can conclude that using an optimal design strategy is not the single, ideal solution to deal with software aging. However, using an optimal design strategy helps to reduce the impact of the symptoms of old age in software products.

6.2 Structural Analysis of the software architecture

In [3], it is described how you can maintain a structural overview of a software project. They make use of metrics like the number of components and the number of calls between components. Also, the call paths between components could be taken into account. They use these types of data from different versions of a product to get an idea of the deterioration of the software products structure from a maintainability point of view. It helps to understand the quality of a software product during its development evolution. However, it seems that a lot of effort is put into creating this analysis and this effort is not directly put in the goal to reduce software aging. Better understanding of the quality of a software product is useful to learn about how software erodes, and eventually may lead researchers to solve the problem of software aging.

6.3 Design for change

An eXtreme programming principal [5] is: "*Welcome changing requirements, even late in development. Agile process harness changes for the customer's competitive advantage.*" In general, the cooperation between the development team and the customers is very good.

In previous sections, we have learned that the causes of software aging generally lead back to changes in the requirements of the software product. Therefore, this principal of welcoming change seems good. Nowadays, we experience that a popular software architecture is the plug-in architecture. Applications like Internet Explorer, Winamp or Eclipse for example allow to be extended with extra functionality from third party developers. There are also numerous other ways to prepare a software product to be so flexible, that changes in the requirements are not major operations.

6.4 Education

The history of software engineering is a very short one in comparison with other engineering disciplines. It is not uncommon for a software engineer to have rolled into the job. The demand for good applications grows and quality attributes are more important than ten years ago. Therefore, it should not be so very surprising that we need better-educated engineers. Someone with an interest to C-compilers can work as a software architect. Not claiming that today's engineers are old people that cannot keep up with today's technology, but more that early generation engineers maybe did too much work in creating technology that we new engineers have a hard time to learn.

Developers often have a high self-esteem and are really proud of their work; they see their work as works of art. This raises a couple of inconveniences. Engineers sometimes have trouble with discarding components which they created themselves. Also, they sometimes deny that there could be bugs in their code. What happens in practice is that people write there own functions, because they did not know that a colleague already implemented the functionality in some library. Also, trust could play a role in this.

In more mature engineering disciplines, an architect or engineer needs to follow proper schooling and has to be certified by a specific association. Considering that we want evolution for our immature engineering discipline, it probably would be a good idea to send out future architects and engineers through proper schooling and certify them to ensure the quality of the software products.

6.5 Documentation

The natural tendency of software developers is that the source code is the best documentation of a software product. And that all effort to write documentation will be in vain, because no one will ever read it. Therefore, it is not a very big surprise that design documents are not always consistent with the source code. It is important that the design documents are clear and up-to-date to prevent architectural drift. Ideally, it should always be relatively easily to find out the reason and rational behind any module or component within the software architecture and good documentation could provide this.

If we take a look at other engineering disciplines, we often see that their design documents are based on mathematical models. As a result, the quality attributes of the final product can be predicted very accurately. Somehow, the idea of using such models seems not feasible for software architecture. Maybe in the future, we might acquire an interest for this way of documenting software products.

6.6 Retirement and death

The last consideration seems grim. Considering that using an optimal design strategy, monitor maintainability, designing to welcome change, educate software engineers properly and writing useful documentation all do not ensure everlasting software products, you might as well better accept the fact that your software product will die of old age.

When you accept that your software product will not last forever, you can plan how long it will take before your software product needs to be replaced. This has the advantage that you do not have the burden of an old piece of software that is hard to maintain. Also, you give customers the feeling that they are being served with new applications. Sometimes the peoples need for new software is rather subjective [2].

Microsoft is an example of a company that works like this. They create new Operating systems at intervals of years. Recent products like Windows XP are published, while older products like Windows 95 are no longer being maintained.

7 Conclusion

By focusing on software aging and its negative consequences, we realize the need for prevention of these consequences. First of all, we discussed what software architecture is in general. After that, we looked at software aging more closely. By doing so, we could identify the causes and corresponding problems regarding software aging. Finally, we proposed an overview of multiple methods for dealing with software aging. However, it appears that software aging is inevitable [1,2]. Nevertheless, the proposed methods reduce the impact of software aging. Sometimes it will be necessary to build a software product from scratch, so all the design decisions can be taken without any revising of other decisions or other implications whatsoever.

8 References

[1] Van Gurp, J., Bosch, J., “Design Erosion: Problems & Causes”.

[2] Parnas, D.L., “Software aging, invited plenary talk”.

[3] Jaktman, C.B., Leaney, J., Liu, M., “Structural Analysis of the Software Architecture – A Maintenance Assessment Case Study”.

[4] Perry, D. E., Wolf, A.L., “Foundations for the Study of Software Architecture”.

[5] Martin, R.C., “Agile Software Development”.

[6] Kruchten, P., “An Ontology of Architectural Design Decisions in Software-Intensive Systems”.

[7] Sommerville, I., “Software Engineering, 6th Edition”.

[8] Kruchten, P., “Architectural Blueprints — The “4+1” View Model of Software Architecture”

[9] Bosch, J., “Design & Use of Software Architectures: Adopting and Evolving a product-line approach”

Software Aging and Design Erosion

D.S.C. Ruiter, K. Werkman
Dept. of Mathematics and Computing Science, University of Groningen
PO Box 800, 9700 AV Groningen, The Netherlands
csg153@wing.rug.nl, k.werkman@student.rug.nl

Abstract. Software aging or design erosion is an unavoidable problem in the lifecycle of every software written. It can be partly prevented and the efforts and financial costs can be diminished to an acceptable level by taking preventive measures in the early stages of the software design process. Good management of and investment into the documentation of the software project is also needed to keep the aging and erosion of software at acceptable levels until the erosion stage has reached a level where redesigning from scratch is preferable above updating.

Keywords: software aging, design erosion, software life cycle, erosion measurement.

1. Introduction

We will begin with a describing the problems with software aging and design erosion in the software life cycle. Then we will discuss the measurability of the design erosion properties. After this several methods to reduce design erosion and software aging are described and finally a short discussion and a conclusion is given.

When time elapses users do expect much more from the programs they use. From one side they would like to use the program with more convenience. On the other side they want to use extra features. These new requirements force the programmer to modify the software to satisfy the user (customer). In spite of the good programming technique of the programmer these modifications will spoil the structure of the software in some way. The spoiled structure of the program due to changing requirements is called design erosion, which can result in decreasing performance.

Requirements have to be changed when the current requirements do not suffice anymore. This can eventually lead to design alterations, which can lead to inefficiency and unclear overview of the working of the software. This is called software erosion or aging.

Requirements do not only change as a result of increasing functionality of the software. It can also occur when new hardware or a new operating system must be integrated in the system. If it is decided to modify the existing software using the same structure, the overall intelligibility of the structure will decrease.

Because of the eroding of the design structure the overall performance will get worse. In general, modifications of the requirements will increase the number of lines in the code. Changing requirements often causes the program to allocate more memory. New code will also bring new bugs. The overall result of many modifications is a program with a feeble performance and a structure that is not understandable for both the original designers and the maintenance engineers.

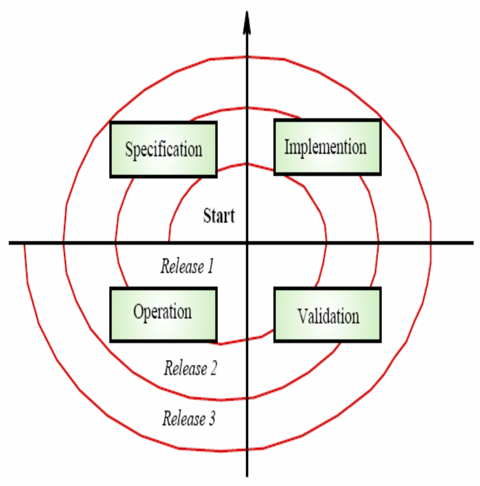


Figure 1: spiral model of the software life cycle

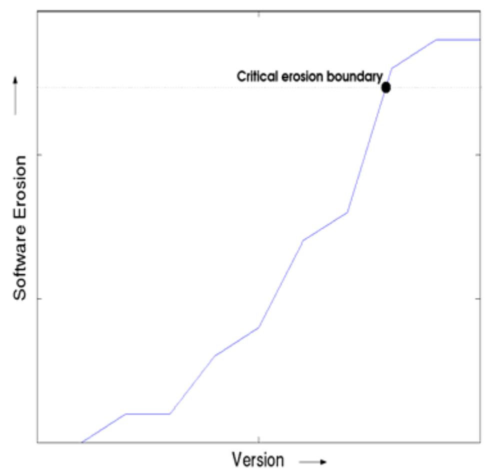


figure 2: graph showing degree of erosion in relation to increasing versions of software product

A software product evolves in time. Before the adjusted and additional requirements can be implemented, these requirements must be specified. The system can be used again after approved testing and validation phases. The spiral model [1] is a good way to represent this continuation of development activities.

The final result over many years with many modifications can be improved (in the next chapter we will discuss preventing design erosion and software aging) by restructuring the code and rewriting some parts of the software. For retaining a good structure it is the best option to rewrite the total software at every change. Due to lack of financial and human resources this is (of course) not possible. The decision whether parts of the software must be rewritten and/or the structure must be improved depends on the current position in the software life cycle [1]. When it is for instance expected that a package will not be used in two years, probably a fast modification of the software will be the best choice.

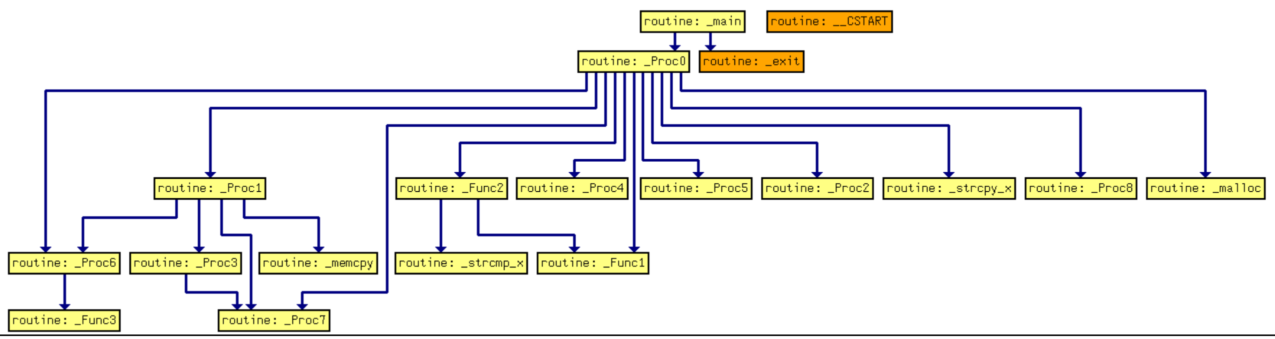


Figure 3: a typical call graph [5]

2. Measureability of design erosion.

Intuitively it is not difficult to know what design erosion is. But when a company must decide to rewrite certain parts of the code, it is preferable this decision can be made on basis of hard numbers. It would be nice if design erosion could be expressed in a figure similar to figure 2, in which the critical erosion boundary is yet to be determined by erosion factors. In each version the ‘design erosion number' is calculated all these numbers can be put in a graph and the managers board know exactly how the design erosion has evolved as a function of a particular version. At a certain degree of erosion the software has reached the point where the maintenance costs and time are no longer acceptable by any standards.

Before the quantification of design erosion is discussed we will assume that the use of a program can be represented by a call graph [2]. This call graph consists of N components (nodes) and E calls (edges). A component can be considered as one single source file. Calls between these files can be drawn as directed edges between the nodes representing the files. A path - with length n - in the described call graph is a set of Edges {Ei ... En} where every edge connects two connected nodes. The level of call graph is the maximum length of a path in the call graph. So four properties in a call graph can be measured.

- **N** : Number Of Components
- **E** : Number Of Calls
- **P** : Number Of Paths
- **L** : Number Of Levels

The measurements can be used to calculate the so called architectural measurements [2]. Table 1 show the measurements that are derived from the before mentioned measurements. These results can be used to plot several figures to indicate the level of design erosion as function of the version number of the software.

Hierarchical complexity	N/L
Structural complexity	E/N
Average Components/path	N/P
Average Paths/Component	P/N

Table 1: Derived Architectural Measures

If the signs of the design erosion are validated by several software developers the design of the software in question can be gradually assigned the status of eroded and maybe drastic decisions have to be made.

3. How to deal with design erosion

Design erosion and software aging exist and one needs to deal with the situation. The software aging can lead to design changes, which leads to design erosion. It may well be that there is no time or willingness in the current software company to take actions to improve the situation due to misunderstanding by lack of knowledge, short term politics regarding the software life cycle or simply not enough financial backup to deal with the problems.

Besides the company culture though, it would be preferable to know if we can do anything to prevent software aging from occurring in future projects.

3.1 Can we prevent software aging?

It is not clear yet if we can prevent software aging. Some researchers say it is impossible to prevent it [3] and they mention 2 major causes for rapid decline in the value of a software product:

- Failure of the products' owners to modify to meet changing needs, which is unpredictable.
- Results of the changes that are made, which result in the software aging causes we already mentioned.

3.1.1 How to prevent software aging

If software aging cannot be prevented, we may be able to partly prevent or delay the aging process. There are so many things to be considered if we want to try to prevent software aging. Let's begin with the design of the software.

3.1.2 Designing software for change

"Designing software for change" is an expression used to indicate that software should be made ready for the future changes, which is nearly impossible.

Many other terms come in mind when taking this subject into consideration, like "information hiding", "abstraction", "separation of concerns", "data hiding", "object orientation" etc. The evolution of new software design principles will continue and henceforth the software made by those design principles will be no better than the model is able to provide. Design erosion follows naturally from the aging stage in the evolution of the design methods. The future changes in software applications are unpredictable and it is impossible to make everything equally easy for change.

Programmers are too eager to get to the first release or to meet deadlines that make them inconsiderate towards programming for future changes. Often management is more concerned with delivering the product in time than future maintenance. Hence programmers are not likely to design for future changes. Also many programmers do not have the appropriate education for the job and are unfamiliar with many design principles and topics like information hiding. Software engineers talk too much and write too many papers about the subject, ignoring what is happening in the fieldwork. "Design for change" is something from the past for them. [3]

Information hiding design is rare to find in software products. The found code is often programmed to just work, often very clever, but rarely designed for future change.

The reason why this is still the case is not that the software engineers do not lack the knowledge, but that they simply do not do it. Programmers tend to think too highly of their own code, that the software they write simply will not have to be changed. This is pure ignorance, since the only programs never changed are the very bad ones nobody wants to use. [3]

3.1.3 Documentation

Documentation should be made for the future maintenance people who are going to read it, not the current writers of the documentation, especially the design principles and decisions (philosophy). Too often programmers say that the code is its own documentation. [3] Mostly the code is written in a specific version of a programming language, hence liable to aging when the language version changes. A good example of this is Visual Basic from version 6 to 7. Version 7 got object oriented design philosophy as a reaction to the success of the Java language and the code in version 6 is simply not compatible without rewriting nearly every class/ procedure/ function. So design principles specific written in version 6 is quite useless when not written in abstracted form.

The reason documentation is not being done properly is that it is not found interesting and has no status in the programming community. Making programs that work is regarded far more important and gives more status to the programmer. Documentation is regarded as the necessary obligatory evil part of the job. The documentation part of the programmers job has to be made very important in the programmers function. But, as stated earlier, often due to deadlines and ignorant management the programmer is not rewarded to write documentation, hence the negative image of documentation continues to exist.

3.1.4 Reviewing

In many other professions, reviewing designs is a very normal part of the job, like designing buildings and ships. There are very precise design prints and documents before the project is even started and it is reviewed very carefully by other experts too. For companies that want their products to last a long time reviewing is a must. However, reviewing in software engineering is known, but not practiced in commercial projects. Some reasons for this are :

- Unqualified programmers with no professional software engineering education.
- Even computer science degree programmers neglect the need for design documentation and reviews, and focus too much on mathematics and science.
- Many “documenters” do not know how to write a readable and precise document.
- Not enough funding for qualified reviewers.
- Time pressure (deadlines) that makes designers think they have no time for proper reviews.
- Programming is regarded as “art” and therefore thinking that others are not fit to or should not review the code.

Designs should be reviewed by others who are responsible for the long-term future of the product. Then maintenance crew should review the code when the design is proposed for the first time too. Sometimes this option is not practical because it is not known if the maintenance crew will be around for that long, as programmers switch often to other projects after the job is done.

3.2 How to deal with changing requirements in existing applications

When software aging has become inevitable and the original designs have been violated the documentation cannot be perfect. Reviewers will miss something eventually and therefore cannot prevent design erosion. In order to deal with changing requirements two strategies can be used. In one strategy the modifications are added in such a way not much time is used. This is called the minimal effort strategy. The optimal design strategy can be applied to create an optimal system. In most application a compromise is used, since it is not possible to use one single strategy.

3.2.1 Minimal effort strategy

Writing updates for versions to meet the demands to keep the software useable and wanted is a very common aspect seeing in nearly all the software applications. But during this process the deteriorating could be slowed down by recreating structure when changes are made in such a way that future changes are made more easily. Documentation and reviewing regarding the design changes should be made to let others know the current state of development. The benefits of this strategy are the relatively low costs.

3.2.2 Optimal design strategy

Sometimes it is more efficient and less time consuming or much cheaper to begin redesign all the necessary components. The design philosophy can be done to meet the needs of the current time. No more time consuming tangling with deprecated and incomprehensible code and documentation. Of course this method is only beneficial when done properly and there is always the chance the same mistakes or even newer ones are made in the new project. If the original design is preferred then the persons involved in the original design could add their knowledge to the project, but this is not always possible for unforeseen reasons. Reusability of working code should be considered since it can be very time saving if it is not tangled up in the code. If the original design has some not yet detected flaw that was part of the cause of the software erosion, the same problem will likely occur in the future and the problem is only delayed, not improves in any way.

3.2.3 Compromise

Because of financial and time limited circumstances it is not always possible to rewrite code completely. In this case one has to reuse some of the old code and rewrite the most urgent parts. Using this strategy will probably not lead to an optimal design.

4. Discussion/Conclusion

We found documentation is the key word in slowing down the software erosion process. In our opinion it would be an option not only to review the design of a software product, but also to review the documentation. If the reviewers find the software is not documented well, the documentation must be adjusted. On the short term this action can be very costly when documentation must be rewritten, but when having a good documentation the maintenance costs will decrease dramatically.

The discussed measurements give an indication of the degree of erosion. But it makes no sense to strictly hang on to the so-called critical erosion boundary. The decision to redesign the software depends on many more factors. For example the current position in the life cycle is an important factor too. The final decision will be made by common sense, but the measurements can back up this decision.

Although we have seen how to try and deal with design erosion, we must acknowledge that is it unavoidable. We will have to deal with the phenomenon sooner or later and can try to reduce the time, effort and costs by optimizing the different phases of the software life cycle. Especially, good design and maintenance documentation of the software projects by educated professionals are mandatory.

5. References

[1] “Software Engineering”, I. Sommerville, 6th Edition, 2001, Addison Wesley, ISBN 0-201-39815-X

[2] “Structural Analysis of the Software Architecture – A Maintenance Assessment Case Study”, Catherine Blake Jaktman, John Leaney & Ming Liu, Computer Systems Engineering, Faculty of Engineering, University of Technology, Sydney, Australia

[3] “Software Aging”, Invited Plenary Talk, D. L. Parnas, Communications Research Laboratory Department of Electrical and Computer Engineering, .htm
McMaster University, Hamilton, Ontario, Canada L8S 4K1, 1994

[4] “Design Erosion: Problems & Causes”, Jilles van Gurp & Jan Bosch Dept. of Mathematics and Computing Science, University of Groningen
PO Box 800, 9700 AV Groningen, The Netherlands
[jilles|Jan.Bosch]@cs.rug.nl,
http://www.cs.rug.nl/Research/SE

[5] “Call graph of the Dhrystone benchmark application for the C16x/ST10 family of microcontrollers “, http://www.aisee.com/graph_of_the_month/aicall

Evaluation of Methods for Area Openings by Connected Set Operators

Marten Pijl (s1277200), Gideon Laugs (s1303104)

Department of Computer Science
RijksUniversiteit Groningen
c/o Blauwborgje 3
Groningen, The Netherlands
gideonlaugs@gmail.com, martenpijl@hotmail.com

Abstract
In this paper, several algorithms related to the field of morphological connected set operators are discussed and compared. Algorithms featured in this paper are Vincent's pixel-queue, Salembier's max-tree as computed by Hesselink and finally, Meijster & Wilkinson's 'union find' algorithm. These algorithm's technical details will be explained briefly. Conclusions are drawn based on analysis of computational complexity and memory usage.

Keywords: Connected Set Operators, Union-Find, Max-Tree, Area Opening, Pattern Recognition, Image Filtering

1 Introduction

Connected set operators (or simply connected operators) form a very interesting class in mathematical morphology. This class of operators is different from other operators in that it operates on connected components rather than individual pixels, as is the case with most operators currently used. As a result hereof, connected set operators possess a number of very useful properties, most important of these the preservation of shape in an image. Connected operators may remove image details completely, but will never remove only part of it and thereby alter the detail's shape.

Important notions in the terminology of connected operators are erosions and dilations. Both these operators work with the aid of a structuring element, which generally consists of a set of foreground pixels in some configuration with some origin. Erosions, basically, work by removing any foreground pixel from an image which neighborhood does not match the structuring element when its origin is placed over said pixel. In contrast, dilations are obtained by placing the structuring element with the origin over all foreground pixels, and adding any foreground pixels in the structuring element to the image. As can be concluded from the above, neither erosions nor dilations are connected operators themselves. However, many connected operators make use of either or both.

Another important notion is that of connected components. Connected components are parts of an image consisting of foreground pixels, which are in some way connected. The most common ways to define connectivity in a two-dimensional image are 4-connectivity and 8-connectivity. In the former case, a foreground pixel is considered to be connected to all foreground pixels immediately to the top, bottom, left and right to it. In the case of 8-connectivity, a foreground pixel is also considered to be connected to any foreground pixel top-left, top-right, bottom-left and bottom-right of it. For the grayscale case, a connected component can be defined thus: two pixels are said to belong to the same connected component if and only if they can be connected by means of a path of constant grayscale value in which every two consecutive pixels are neighbors with respect to 4-connectivity or 8-connectivity, depending on the chosen connectivity. The general idea behind connected components is illustrated in (Fig. 1).

The earliest connected operators known in literature were the opening by reconstruction for binary images, and the corresponding closing by reconstruction. The opening by reconstruction works by performing an erosion with some

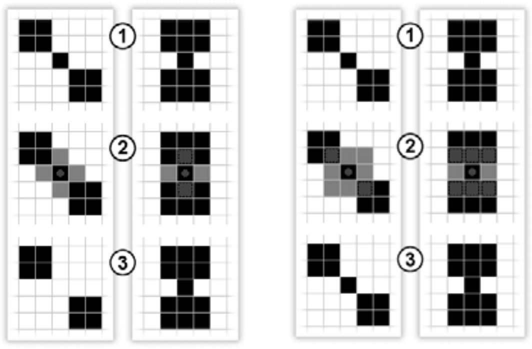


Fig. 1: A visual explanation of the connected component concept. In block A is shown what 4-connectivity means with regard to an image. In block B the same image is used again, this time using 8-connectivity. In both blocks, three steps are done. Step 1 is the original image. In step 2, the concept of 4- and 8-connectivity is applied. Step 3 then shows the resulting connected components using either 4- or 8-connectivity. When comparing the leftmost columns of both block A and B, one can clearly see the difference 4- and 8 connectivity can make. Using 4-connectivity, two separate components are created, whereas when using 8-connectivity, the two components are considered connected and thus one components instead of two.

structuring element, and then restoring all foreground components not completely removed by the erosion. At a later stage, more connected operators have been devised, such as area openings, dynamics filters, and complexity, volumetric, motion and motion-oriented operators. These operators provide additional selection criteria, such as selection on shape, contrast, or size, for example.

Apart from functioning on binary images, these operators can also be adjusted to function on gray-scale images. Rather than using strictly connected components (i.e. where all pixels have the exact same gray-level), flat zones are generally used. A flat zone is different from the basic definition of a connected component in that it may be composed of pixels with a low gray-level fluctuation. This means that in the case of flat zones, image details are considered connected as long as the change in gray-level is a subtle one. Converting an image to flat zones is sometimes known as soft binarization. The 'softer' the binarization, the fiercer gray-level transitions may be. In contrast, a very 'hard' binarization is similar to using classic connected components. Flat zones are generally useful in realistic images, where small fluctuations in gray-level are very common, and noise is generally a factor. In such a case, using connected components would yield a very large number of very small connected components.

Area openings and closings:

Area openings and closings have proven to be an important development for connected operators, finding their uses in a multitude of pattern recognition applications, as well as proving useful in image processing applications, for instance, the removal of background noise from an image. In this paper, several algorithms for computing them will be described later on. First, a description of these connected operators may be in order. In the simpler binary case, area openings remove all connected foreground components with an area smaller than some threshold λ. This means that area openings (as well as closings) have component area as their selection criteria (as opposed to, for example, openings by reconstruction, which have shape as their selection criteria). Area openings and closings have been defined thus, according to [1]:

Definition 1: Let $X \subseteq M$ and $\lambda \geq 0$, where M is the domain of the image X. The binary area opening of image X with scale parameter λ (the minimum area size allowed) is given by:

$$\Gamma_\lambda^a(X) = \{x \in X \mid A(\Gamma_x(X)) \geq \lambda\}$$

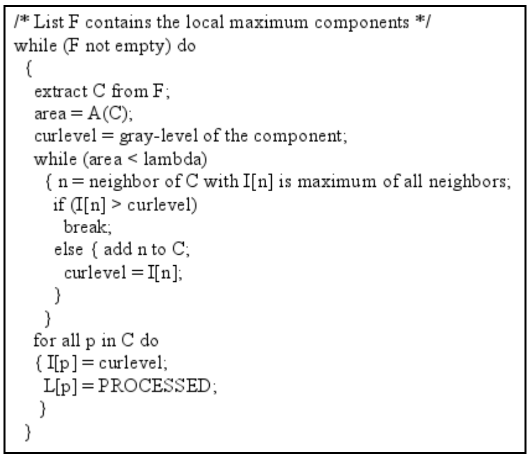
Here, the function A defines the combined area of the pixels. The binary area closing can be defined by duality, where C is used to indicate the complement

$$\Phi_\lambda^a(X) = [\Gamma_\lambda^a(X^C)]^C$$

The definition of an area opening of a gray-scale image f is usually derived from binary images $T_h(f)$ obtained by thresholding f at h. These are defined as

$$T_h(f) = \{x \in M \mid f(x) \geq h\}$$

At a later date, area openings and closings have been extended to a greater class of attribute openings and closings, as well as thickenings and thinnings. These extended connected operators can use any size property as selection criteria, not just area (such properties may include moment of inertia or smallest diagonal, for instance). However, the algorithms in this paper will be restricted to the traditional area openings and closings.



```
/* List F contains the local maximum components */
while (F not empty) do
  {
   extract C from F;
   area = A(C);
   curlevel = gray-level of the component;
   while (area < lambda)
     { n = neighbor of C with I[n] is maximum of all neighbors;
       if (I[n] > curlevel)
         break;
       else { add n to C;
         curlevel = I[n];
       }
     }
   for all p in C do
    { I[p] = curlevel;
      L[p] = PROCESSED;
    }
  }
```

Fig. 2: Pseudocode of the core of Vincent's area opening algorithm. The parameter lambda in the pseudocode represents the area threshold λ.

2 Explanation of techniques

2.1 The Pixel-Queue algorithm

The pixel-queue algorithm described below is the first of several algorithms described in this paper, which are able to perform area opening. First, it is important to define several conventions. First off, a flat zone L_h at intensity level h of some gray-scale image I is defined as a connected component of the pixel set $\{p \in M \mid I(p) = h\}$. A regional maximum M_h is a component such that it contains no members, which have a neighbor with a grayscale value larger than h. A peak component P_h is a connected component of the thresholded image $T_h(I)$ at level h. Note that there may be more such components per level. In this case, they will be indexed like P_h^j, indicating a component at level h with some index j. Also note that any regional component is also a peak component, though the reverse is not necessarily true. An impression of the algorithm is alternatively provided by some pseudocode in (Fig. 2).

The algorithm works by first constructing an array of all regional maxima in the image. Next, all regional maxima acquired this way are processed individually to form a peak component with an intensity level at least equal to that of the corresponding regional maximum. This is done by taking a random pixel within the regional maxima, and then expanding it to include the remaining pixels of the peak component.

To do this, a queue is created, containing all neighbor pixels of the random point. Of this queue, the pixel with the highest intensity value is obtained. If the intensity level of the pixel is not higher than the level of the last pixel (more on this later), it is added to the component, and any yet to be processed neighbors are added to the pixel queue. Obviously, once processed, the pixel is removed from the queue. Again, the highest level pixel is chosen from the queue.

This process continues until the total number of pixels equals or exceeds the minimum area size λ, or until a pixel is found with an intensity level higher than the level of the last selected pixel. In either case, the intensity level of all pixels in the component is set to that of the last pixel processed, and the regional maximum is removed from the list of regional maxima yet to be processed, and the next regional maximum is chosen to be processed.

So what does a round in this algorithm accomplish? There are two cases in which such a round may terminate. First, the case where the total pixel area of the component starts to exceed λ. In this event, another two cases are possible. If the peak component had an area larger than or equal to λ, all pixels processed must be of the same level as the peak component, since there are no pixels with a higher intensity level in or neighboring the level component, by definition of a regional maximum. Also, the algorithm always picks the pixels with the highest level available. This means the level of all processed pixels remains the same, so the component is unaltered.

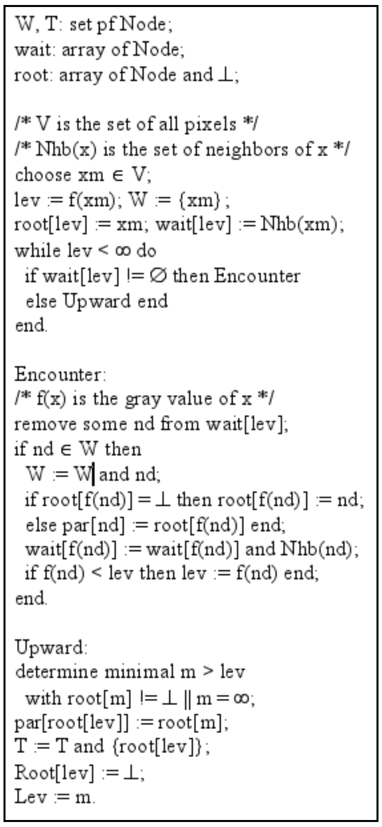
However, in the case where the peak component had an area less than λ, some pixels of an intensity level less than that of the peak component must have been selected to reach the required number of λ pixels. Also, any pixels processed later cannot have an intensity level which rises again, because of the second termination condition. This makes sense, as pixels with a higher intensity level than the last would have to come from a different peak component (all pixels belonging to the original component were selected first). This means that, as the level of all pixels in the component is set to the level of the last component, the peak component is deleted.

Note that this means that several 'layers' may be removed in the same step. If the area of the peak component, plus that of the underlying component is still insufficient, both may be removed.

Another cause for termination of the processing step is the case where a pixel is found with an intensity level higher than the last pixel processed. In this case, another peak component has been encountered. This means further processing now is not needed. All pixels in the component are set to the appropriate level, and the next regional maximum is selected. Note that it is possible that the region is processed once more when the encountered component is itself evaluated, however.

2.2 The Max-Tree approach

Unlike the pixel-queue, a max-tree is a data-structure rather than an algorithm. However, once a max-tree is constructed, performing an opening is nearly trivial. Therefore, most of the effort consists of creating the max-tree. For a pseudocode impression of the algorithm, refer to (Fig. 3).



```
W, T: set pf Node;
wait: array of Node;
root: array of Node and ⊥;

/* V is the set of all pixels */
/* Nhb(x) is the set of neighbors of x */
choose xm ∈ V;
lev := f(xm); W := {xm} ;
root[lev] := xm; wait[lev] := Nhb(xm);
while lev < ∞ do
  if wait[lev] != ∅ then Encounter
  else Upward end
end.

Encounter:
/* f(x) is the gray value of x */
remove some nd from wait[lev];
if nd ∈ W then
  W := W and nd;
  if root[f(nd)] = ⊥ then root[f(nd)] := nd;
  else par[nd] := root[f(nd)] end;
  wait[f(nd)] := wait[f(nd)] and Nhb(nd);
  if f(nd) < lev then lev := f(nd) end;
end.

Upward:
determine minimal m > lev
  with root[m] != ⊥ || m = ∞;
par[root[lev]] := root[m];
T := T and {root[lev]};
Root[lev] := ⊥;
Lev := m.
```

Fig. 3: Pseudocode of the max-tree algorithm

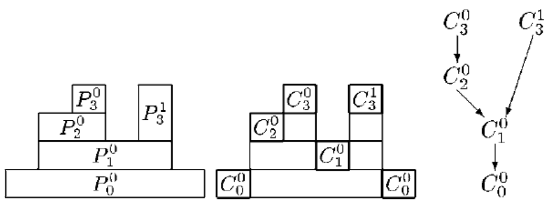


Fig. 4: A visual explanation of the max tree structure

The max-tree (which has a dual, called the min-tree) takes the shape of a tree with each of its nodes C_h^j corresponding to a peak component P_h^j at a certain threshold level h, where the node only contains pixels of gray-level h. In addition, each node except the root points to a parent with an intensity level lower than the node. The root of the tree is formed by the background of the image. It is also important to note that not all nodes at every gray-level need be occupied. Perhaps the best way to come to terms with the concept is by means of an illustration: see (Fig. 4).

Once the tree is constructed, performing an area opening is simply a matter of checking each node's pixel area against the required threshold λ. If the area equals or exceeds λ, nothing needs to be done. Otherwise, the node is simply removed, and its pixels are merged into the parent node. One of the disadvantages of the max-tree approach is that problems may occur when some node is accepted, but its parent rejected. Fortunately, this will not occur in this case: if a node has large enough area, any parent node (which must enclose the pixels of said node) will also have sufficient area.

To compute the max-tree, the breadth-first approach devised by Hesselink [3] can be used. The algorithm works by picking a random pixel from the image, which is marked as processed, and starts working from there. The algorithm iterates itself until the root (where the gray-level equals an arbitrary -1) is reached. Like pixel-queue, the algorithm defines a queue containing pixels, from which it continues to draw until it is emptied (unlike pixel-queue, the ordering is chosen randomly). Initially, this queue is filled with the neighbors of the initial pixel. Whilst there are still pixels in the queue, the algorithm calls 'encounter', otherwise it calls 'down'. On a final note, the queue is gray-level specific: every gray-level has its own queue.

In the case of an 'encounter', an unprocessed pixel (previously processed pixels are ignored) are checked for any corresponding component of the same gray-level. This is done by an array storing the first pixel encountered at that gray-level. If no such component exists, it is created. Otherwise, the pixel is added to the component. In either case, the pixel's neighbors are added to the queue. Then comes the tricky bit. If the chosen pixel has a gray-level higher than the current component, the new gray-level will be processed first. The old component is abandoned for now, along with its queue, the new queue just containing the pixel's neighbors. Note that the algorithm will not switch to a lower gray-level: it will create a queue and necessary components, though.

Once all pixels in the queue of the current gray-level have been exhausted, the algorithm calls the 'down' procedure. Here, the largest possible gray-level smaller than the current component's is selected, such that a component is known, or the gray-level is –1, the 'root' of the whole image (this won't happen unless all pixels have been processed). Then, the new component is set as the parent of the previous component. Also, with the previous component completed, a possible new component of that gray-level can be selected.

It may require several reads to understand this fairly complex algorithm. In basic terms, it starts at some pixel, and tries to find a local maximum. Once it has found this, it gathers all pixels, which form a part of it, and labels it as a component. Next, it tries to find the parent of this component, and complete it, moving down the gray-level scale. This continues until it has processed all pixels, or until it encounters a new local maximum, in which case it starts to pursue this, before tracking back down again.

```
void MakeSet (int x)
{ parent[x] = -1;
}

int FindRoot (int x)
{ if (parent[x] >= 0)
  { parent[x] = FindRoot(parent[x]);
    return parent[x];
  }
  else return x;
}

boolean Criterion (in x, int y)
{ return ( (I[x] == I[y]) || (-parent[x] < lambda) );
}

void Union (int n, int p)
{ int r = FindRoot(n);
  if (r != p)
  { if (Criterion(r, p))
    { parent[p] = parent[p] + parent[r];
      parent[r] = p;
    }
  else
    parent[p] = -lambda;
  }
}
```

Fig. 5: Pseudocode for the basic operations of area openings and closings

```
/* array S contains sorted pixel list */
for (p=01 p<Length(S); p++)
  {
    pix = S[p];
    MakeSet(pix);
    For all neighbors nb of pix do
      if ((I[pix] < I[nb]) ||
          ((I[pix] == I[nb]) && (nb<pix)))
        Union(nb, pix);
  }
/* resolving phase in reverse sort order */
for (p=Length(S)-1; p>=0; p--)
  {
    pix = S[p];
    if (parent[pix] >= 0)
      parent[pix] = parent[parent[pix]];
    else
      parent[pix] = I[pix];
  }
```

Fig. 6: Pseudocode showing how to perform an area opening using the operations of (Fig. 5).

2.3 The Union-Find algorithm

The union-find algorithm works slightly different from the previous two described algorithms, in that it is able to process multiple peak components at a time. These peak components are grown until the required area size λ is reached, new components being created and merged along the way. Pseudocode of this algorithm is given in (Fig. 5) and (Fig. 6).

The algorithm's heart consists of two data-structures, the first of which being an array containing all pixels in the image (unsorted). This data-structure is named 'parent', and actually combines two types of data. In case a positive value is found, the value refers to the pixel which is nominated as the root of the component which the pixel is part of. If a negative value is found, this indicates that the pixel is a root itself, and the value represents the area of the component discovered so far (though the minus sign obviously needs to be ignored in this case). The second data-structure is an ordered queue containing all pixels in the image. The queue is ordered on gray-level, and any pixels of the same gray-level intensity are ordered in a scan-line fashion. This means ordered on lowest y-coordinate first, and then on lowest x-coordinate, in a sense traversing the image horizontally from the top-left to the bottom-right.

The algorithm works in two steps. First, it determines the connected components in the image. Once this has been done, the image intensities will be re-evaluated. For the first part of the algorithm, the pixels are sequentially removed from the queue, and processed individually. First, a new set is created with the pixel as its only member, and its 'parent' value is set to –1 (this represents a component of a single pixel, with area 1). Then, all neighbors of the pixel are evaluated. All of these which have been already processed themselves are tested to see if they need to be merged with the newly created set. This is attempted if the intensity of the currently processed pixel is less than the intensity of the neighbor in question, or when both the intensities match and neighbor's area exceeds or equals the current pixel's area.

To perform this merging, the root of the neighboring pixel is determined first (the processed pixel is its own root). If the roots match, there is no need to merge the sets. Otherwise, yet another, final criterion is imposed. If the pixel intensity of the pixel and the neighbor's root match, or if the area of the neighbor's root is still less than λ, the sets are merged. Note that in the first case, the sets belong together. However, in the second case (when the first is false), the neighbor's component is removed, as will become clear later. This is because the current pixel must have a lower intensity than its neighbor's component (because of the previous tests), and the neighbor's component already includes all pixels of the same intensity (because of the way the queue is ordered).

When the sets are merged, the processed pixel is taken as the new root, and the new area is computed. The ‘parent’ of the neighbor is set to the processed pixel. However, if the final criterion is false, the sets are not merged. Instead, the ‘parent’ of the currently processed pixel is set to – λ, as it is known that there is a neighbor with a higher gray-level intensity, and that it has a sufficiently large area. This must mean that the current pixel can be accepted as well. In any of the other cases, no action is taken.

As mentioned, in the second step of the algorithm the intensities of the pixels are recomputed. This is done by traversing the queue in reverse order, and by investigating the ‘parent’ attribute of each pixel. If the pixel is not a root (in other words, it has a value greater or equal to 0), the intensity is set to the intensity value of its root. It is important here to remember that because of the order in which both steps are executed, a root value will always be evaluated before any other members of it’s component.

3 Comparison of techniques

In the previous section, several techniques have been discussed. In this section, the focus will shift more towards the differences and similarities between these techniques, as well as their computing complexity.

3.1 Technical Comparisons

First of all, of the techniques discussed, one is clearly distinct from the other three: the pixel-queue algorithm. The other three algorithms all can be seen as tree-based algorithms. Both the union-find and the breadth-first algorithms are mere variations and improvements of Salembier’s max-tree algorithm.

Historically speaking, the pixel-queue algorithm is the oldest algorithm to be discussed here, introduced by L. Vincent in 1993 [4] [5]. What makes this algorithm significantly different to the other three algorithms discussed above is that instead of using a tree-based approach to compute area openings and closings, the pixel-queue algorithm uses a technique based on a priority queue to obtain a solution. The specific details to this technique have been discussed in section 2, so there is no need to reproduce those here. However, it is important to note that the use of a priority queue and the ordering with which the pixels are retrieved from it, result in large complexity. This will be discussed in more detail in section 3.2.

Although the priority queue based approach clearly sets the pixel-queue algorithm apart from the other three algorithms, there are two key properties shared by both the pixel-queue as well as the max-tree algorithms. First, both algorithms use a method based on flooding. Second, the two algorithms process an image one peak component at a time. Although the max-tree algorithm proved much more interesting than the pixel-queue algorithm, its major downside was in the latter characteristic shared with the pixel-queue algorithm. In order to resolve this, Meijster & Wilkinson devised a slightly modified version of the max-tree algorithm, called the union-find method [1]. Tarjan’s union-find algorithm [6] for keeping track of disjoined sets is incorporated into Salembier’s max-tree algorithm, which enables the method devised by Meijster & Wilkinson to handle multiple peak components simultaneously by creating and merging peak components as needed while keeping track of their area. Merging peak components and thereby increasing its area ceases as soon as a certain area threshold λ is reached, after which the merged peak components can be handled as one. As will be shown in Section 3.2, this modification did not do the time complexity any good, but did improve its usability by eliminating the need to handle each peak component individually.

Another modification to Salembier’s max-tree construction algorithm is devised by Hesselink, in which breadth-first search is incorporated to simplify the original structure [3]. This breadth-first version characteristically differs from other versions of the max-tree construction algorithms in the way it builds up the tree structure. Whereas Salembier uses a recursive procedure containing three nested loops to set up the tree structure, Hesselink creates the tree in a single repetition. For the technical details to this algorithm, as well as those behind the union-find modification, please refer to sections 2.2 and 2.3 respectively.

3.2 Efficiency Comparisons

After having shortly evaluated the main differences in characteristics between the four algorithms described in section 2, we can focus on the difference in complexity concerning these algorithms.

Whenever several different algorithms are all suitable for performing a certain task, it is primarily the efficiency with which an algorithm performs this task that is the key factor. Although usually efficiency scales are defined through analysis of the number of operations needed in a worst-case scenario, there are situations in which this technique is not capable of delivering a suitable decision factor. Such a situation is encountered when regarding algorithms for set openings and closings, as will be shown later in this section.

Meijster & Wilkinson did extensive research to the time complexity and computational behavior of the pixel-queue, the max-tree and the union-find algorithms [1]. All three algorithms were tested on both synthetic images as well as natural images.

With regard to the synthetic images, four different types were used: one pair of images focusing on the difference in image content and another pair focusing on image size. The results of these tests are graphically shown in (Fig. 7).

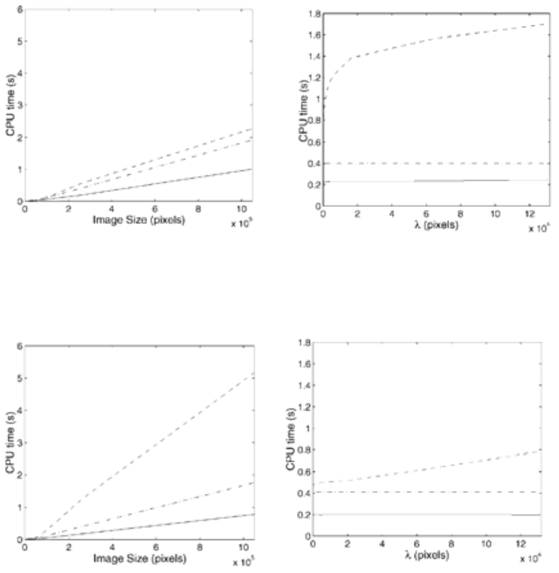


Fig. 7: Graphical display of results obtained from testing the pixel queue (dashed), max-tree (dash-dot) and union-find (solid) algorithm's performance. The upper two graphs show the timing results for distance maps, whereas the lower two graphs show the dependence of computing time on the image size.

From the top two graphs in (Fig. 7), it can be clearly seen that the pixel-queue algorithm is heavily dependent on both the image content as well as the λ. In contrast, the union-find and max-tree algorithms appear to be independent of λ and to have only a minor dependency on the image content. When looking at the lower two graphs in (Fig. 7), it shows that only the union-find and the max-tree algorithms are independent of the density of local minima, but all three algorithms are linearly dependent on the image size.

The above conclusions are drawn from practical experiments. But how do these algorithms compare to one another on a theoretical basis? First of all, characteristic to the complexity of the pixel-queue algorithms is the use of a priority queue. Accessing a pixel in a priority queue takes O(logN) computational complexity.

Apart from this, a worst-case scenario has to be considered, in which λ is chosen in such a way, that only the entire image (consisting of N pixels) satisfies the criterion. As shown by Wilkinson & Roerdink [7], the worst-case computational complexity of the pixel-queue algorithm becomes O(N^2logN).

As far as the max-tree algorithm is concerned, it is important to realize that the computational complexity is dominated by its most complex component. For the max-tree algorithm this is the flood filling process, which - as experimentally shown above - is linear with respect to the connectivity and in the number of pixels. The algorithm has a worst-case complexity of O(N).

The same also holds for the breadth-first version of the max-tree algorithms [3]. Although the removal of the recursive procedure makes the breadth-first version a lot simpler than its original, the algorithm is still dominated by its most complex part, the flood filling process. As has been discussed above, flood filling has a worst-case complexity of O(N). Thus, the worst-case complexity of the breadth-first modification is still O(N).

Finally, the complexity of the union-find method can be assessed in a similar way. Most important is the argument that it is the most complex component of an algorithm, which dominates and thus defines the entire algorithms complexity. Now, assuming that the most complex part of the union-find method is the union-find itself, i.e. the construction of the tree, Meijster & Wilkinson state that although less complex algorithms could be used, the best algorithm to use in the union-find method is of complexity O(NlogN). Moreover, Tarjan [6] provided theoretical backup to the above assessment by deriving a worst-case complexity of O(NlogN) in situations like the union-find algorithm.

One of the most obvious conclusions that can be drawn from the complexity analysis above is that the pixel-queue algorithm is deemed obsolete compared to the efficiency achieved by tree-based algorithms. Next on, one's attention is easily drawn towards the difference in complexity between Salembier's max-tree algorithm and the union-find modification of it by Meijster & Wilkinson. From a historical perspective, it is easily seen that although the union-find algorithm is of a more recent release, it actually performs worse than its original. One could be easily tempted to conclude that devising the union-find modification was unnecessary due to its non-improvement in complexity. However, instead of just focusing on the computational complexity, the union-find algorithm is a case in which special attention has to be paid to other factors, primarily being the memory usage. From analyses done by Meijster & Wilkinson [1], it appears the union-find algorithm uses a significantly less amount of memory than the original max-tree algorithm. In fact, using a three-dimensional test volume measuring

128*128*62, the difference in memory usage was experimentally shown. The max-tree algorithm performed worst with 57.5 MB and second came the pixel-queue algorithm using 45 MB. The union-find algorithm needed less than half the memory needed for the max-tree algorithm, with only 25 MB. Now one can only imagine the difference this makes when considering enormous datasets.

4 Conclusions

All algorithms discussed in this paper all have their advantages as well as their disadvantages. Be it their ease of implementation or their computational complexity, in one way or another each of these algorithms once proved or still proves to be of significant importance to the are of pattern analysis. However, as time passes new ideas rise. With regard to the algorithms related to connected set openings and closings, Salembier's idea of using a tree-based approach made for significant improvement in computational complexity. But the creation of the union-find and the breadth-first modifications proved it could still be done better.

As far as the algorithms discussed in this paper are considered, Meijster & Wilkinson state that the union-find algorithms outperform the pixel-queue and the max-tree algorithms. As shown in section 3.2, the complexity of the breadth-first algorithm is equal to that of the max-tree, implying the union-find method also outperforms the breadth-first algorithm. Although situations might exist in which the three more complex algorithms would be a more sensible choice than the union-find algorithm, judging from their performance compared to that of the union-find, the pixel-queue, max-tree and breadth-first algorithms can generally be regarded as being rendered obsolete by the union-find algorithm.

5 References

[1] A. Meijster, M.H.F. Wilkinson, *A comparison of algorithms for connected set openings and closings*, IEEE Trans. Pattern Analysis and Machine Intelligence (2002)

[2] P. Salembier, A. Oliveras, L. Garrido, *Anti-extensive connected operators for image and sequence processing*, IEEE Transactions on Image Processing (1998)

[3] W.H. Hesselink, *Salembier's min-tree algorithm turned into breadth first search*, Elsevier Information Processing Letters (2003)

[4] L. Vincent, *Morphological area openings and closings for grayscale images*, Shape in Picture: Mathematical Description of Shape in Grey-level images, NATO (1992)

[5] L. Vincent, *Grayscale area openings and closings: their efficient implementation and applications*, Proc. EURASIP Workshop on Mathematical Morphology and its Application to Signal Processing (1993)

[6] R.E. Tarjan, *Efficiency of a good but not linear set union algorithm*, Journal of the ACM (1975)

[7] M.H.F. Wilkinson, J.B.T.M. Roerdink *Fast morphological attribute operations using Tarjan's union-find algorithm*, Proc. Int'l Symp. Memory Management 2000 (2000)

[8] M.H.F. Wilkinson, *Connectivity preserving filters in morphological image analysis*, Undergraduate Student Colloquium, Institute for Mathematics and Computing Science, University of Groningen (02-12-2004)

A look at Programming Methods for solving problems of current Software Development

Kenneth Rohde Christiansen, Niek Oost
k.r.christiansen@student.rug.nl, h.oost@student.rug.nl

Department of Mathematics and Computing Science
Rijksuniversiteit Groningen
Blauwborgje 3
NL-9747 AC Groningen

Abstract
In this paper we will look at the problems of application development that are not directly solved by using standard programming methods like imperative, object-oriented or functional programming. We will introduce four proposals for alternative programming methods developed to solve some of these problems. This text evaluates the different proposals on practical applicability of these methods in the near future. The document assumes that the reader has basic knowledge of object-oriented programming.

Keywords: Programming Methods, Subject-Oriented Programming, Aspect-Oriented Programming, Separation of Concern, Computer Science.

Version of Februari 27th 2005; revised form of January 9th 2005.

1 Introduction

The main objective of software engineering is to improve software quality, reduce costs and also to facilitate easy maintenance of the software product. Over the last few years many new techniques have been introduced to support this. High-level languages such as C proved to be a major step forward from programming in assembly and today most people value the use of Object-Oriented languages with garbage collectors such as Java or C#. Unfortunately, Object-Oriented languages do not solve all of the software development problems.

The main objective of this paper is to look into the various problems of application development that are not solved by common programming languages in use today. The biggest problems of software development relates to code understanding, maintenance, extendibility and reuse. Object-Oriented Programming has been suggested to solve some of these problems by decomposing the code into small reusable objects, and has proven highly successful. Unfortunately, artifacts as for instance new features often decompose the product in a different way, so the different decomposition methods tend to scatter or tangle the code. Adding a new feature might require changes to multiple objects, this will make it harder to understand and maintain the code. The different ways a product can be decomposed are often referred to as the separations of concern. Each concern leads to a different kind of separation/decomposition.

1.1 Discussed techniques

In this text the following proposed techniques are discussed :

Subject-Oriented Programming [Harrison and Ossher, 1993] tries decomposing the code into subjects that deal with the same objects, separating the extrinsic properties from the actual objects and their intrinsic properties.

Aspect-Oriented Programming [Kiczales..., June 1997] looks at the non-functional concerns to be separated from the functional concerns.

Hyper modules and -slices are introduced in [Harrison..., 1999] as a general way to satisfy the separation of concern for functional dependencies.

Superimposition [Bosch, 1998] proposes a technique for component adaptation for reusable components.

2 Subject-Oriented Programming

Object-Oriented programming tries modeling the world as objects. Trees, birds, cars, etc. are all examples of objects. This has proven to be a successful approach and has revolutionized the software industry. Unfortunately, different subjects consider different properties of an object as important. Where a tree has intrinsic properties like height and age, a subject can also be interested in other properties. A bird, for instance is also interested in the possibilities to build a nest in the tree.

In the software developing community a growing suite of applications manipulate the same objects. During the development of these objects, programmers need to anticipate which properties that could be used in future applications, so that it is easier to reuse their code. This is an almost impossible task, since it is almost impossible to know what extrinsic values will be used in future applications. An extensive interface on the other hand also creates problems for programmers using these objects: different applications will only use part of the object interface, while they are forced to also consider the rest of the interface (with possibly state modifiers).

If object-oriented programming is to scale from the development of independent applications to the development of highly integrated application suites it has to relax the dependency of objects and instead concentrate more on how these objects are tied together. A technique that emphasizes on the subject view is known under the name Subject-Oriented Programming and is described in [Harrison and Ossher, 1993]

2.1 The idea

Subject-Oriented Programming builds on Object-Oriented programming, in that there exist objects containing intrinsic properties (often known as fields) and behavior (often known as methods). Each subject contains a library of the classes of objects known to the subject. The subject additionally contains extrinsic properties for each of these objects reflecting how the subject sees the given object.

By following this idea it is possible to develop subjects independent of other subjects, while they can still deal with the same objects. An application is then considered as a subject or a composition of subjects.

2.2 Subject composition

When a subject interacts with an object it might influence the objects' intrinsic properties, but this might, in effect, also influence the extrinsic properties of different other subjects. If, for instance, a bird subject builds a nest in a tree this might mean that the woodman is not allowed to cut down the tree, thus the `can-cut` property owned by the woodman has to be changed.

To make a subjects' action on an object affect the extrinsic properties of another subject, the subjects need to be composed. This can only work if a subject implements the same behavior/methods as the behavior of the subject that it wants to respond to. If the woodman wants to track the building of nests in a tree, which are controlled by the `build-nest` and `abandon-nest` behavior of the bird subject, the woodman subject will have to implement these as well.

2.3 Class matching

The subjects do not necessarily have to include the same class library. For instance, a pine might be derived from `object->nestable` in the bird subject or it can be derived from `object->tree->softwood` in the woodman subject. This means that the composition rule also needs to specify some kind of class matching. Additionally, some subjects might know objects that other subjects don't know about. For instance the bird might know about a tree (a locust for instance) that the woodman doesn't know about. The composition rule can then be used for concluding that a locust can be treated by the woodman as being a tree.

2.4 Subject activation

When a subject gets instantiated - or activated - to follow the terminology in the paper, all objects in the subject have to be created as well as the extra extrinsic properties reflecting the subjects view of the objects. In a subject composition there is more than one subject dealing with the same objects. This means that the other subjects need to instantiate their extrinsic properties for the objects as well. This can be done in different ways, for instance all extrinsic properties for all subjects will be instantiated when one of the subjects is activated. A better and more effective way may be to first instantiate these properties when needed.

2.5 Our Conclusion

Subject-Oriented Programming introduces some nice concepts for modeling the interaction of subjects in the real world. Whether this would work for modeling applications is a good question. We think it sounds very complicated to design - maybe huge - different class libraries for each subject. The idea is that these can be designed independently, but for class matching to work properly the developer probably needs to know about the class libraries in the other subjects. Another problem that we see, is the problem of saving/serializing objects. Lets consider a document to be an object and a word processor and a spreadsheet to be subjects implementing extrinsic properties for this document. If we now want to save the document the question is how we make sure that all extrinsic properties for all subjects are saved. We don't see an easy way to do this with the current model.

3 Aspect-Oriented Programming

When code has to be optimized (for example for performance), these optimizations often cross-cut the different components that the application consists of. During the optimization of a piece of code, one often reduces the number of procedure calls, creating code that is much harder to understand and maintain, than the original code, but faster. To get the best of two worlds : easy maintainable code and highly optimized software, aspect oriented programming is proposed. In this technique, 'aspects' are introduced as issues cross-cutting components, dealing with things like performance and memory usage.

The paper [Kiczales..., June 1997] uses a good example to show what Aspect-Oriented Programming is all about. When dealing with images, like for instance developing an OCR application, you often have to put the image through various filters. Each of these filters produces a new image which might live shortly before it is copied to a second filter, discarding the copy of the previous filter. Since this results in excessive memory references, which can result in cache misses, page faults, etc. it should be avoided. In this particular case developers often try to merge these different filters into one big procedure, reusing as much memory as possible. This merge is done by hand, and often results in buggy, tangled code that is hard to understand, hard to extend and hard to maintain. The authors of the paper implemented an ineffective subpart of an OCR application in as little as 768 lines of code. The effective, tangled version on the other hand consisted of 35213 lines of hard maintainable code.

3.1 How it works

In the example described earlier, the optimization is done by merging filters, which is in fact done by merging loops. Aspect-Oriented Programming can do this automatically, and does it by introducing a component language, an aspect language and an aspect weaver. It is also noted that the actual weaving can be done at runtime or at compile-time.

The idea is that you write your components in a language similar to what you are used to, but for the above example, the language will have to support a high-level looping construct so that the weaver can detect, analyze and fuse loops more easily. In the paper they introduce a `pixelwise` construct to do this.

Additionally, you will have to write a merging condition in the aspect language, which will then be used to compare nodes in the data flow of the application. If two of the filters fulfill the condition, like for instance that they are both pixel wise, then they can be merged. This exact merging is done by the weaver.

The weaver uses unfolding to generate the data flow graph, which then is processed by the aspect rules which will look for nodes to merge. As the last step the code generated walks though the data flow graph and generates the actual code.

With this method the authors implemented an easy maintainable version of the OCR subroutine, that was almost as effective as the tangled version and in as little as 1039 lines of code. With an improved code generator it should be possible to make the new version practically as effective as the tangled version.

3.2 Our Conclusion

There are many different kinds of aspects, that cannot cleanly be encapsulated in a generalized component. Loop fusion is just one of these. Others include, minimizing network traffic, synchronization constrains, error handling etc.

We find Aspect-Oriented programming to be an interesting new idea that can actually easily solve some of the problems that developers are dealing with today. Unfortunately, Aspect-Oriented programming is a very young idea and there has to be researched how it can help solving other aspects than loop-fusion, and what are good structures for use in aspect programs. There is also the question if people can easily learn to analyze and identify aspects in their programs, and if it can be made easy to debug these applications when the generated code will be very different from the actual implementation.

4 Hyper modules and -slices

Hyper modules and hyper slices is a recent idea for solving the separation of concern. It builds on Subject-Oriented Programming and the people behind Subject-Oriented Programming are also co-authors of this paper, [Harrison..., 1999].

The idea roots in Object-Oriented programming, and facilitated methods for easily adding new features that span multiple classes. A hyper slice resembles a subject in the Subject-Oriented method. The difference is that a hyper slice only implements extra 'extrinsic' properties and methods for the classes in the program. This means that we don't have the class matching problems as in the Subject-Oriented idea. This makes it possible to easily add new features without changing and polluting the existing classes with implementation details of this feature. This way the code for the feature addition will be kept together and code readability and maintainability will be maintained.

As with subjects, hyper slices can be combined by a composition rule. A composition of hyper slices is called a hyperplane, and it contains approximately the same problems as we noted in the section about Subject-Oriented Programming. There are fewer problems, as the slices build on top of an already defined class library. Because the slices do not implement different class libraries, class matching is much easier than with Subject Oriented programming.

4.1 Our Conclusion

Hyper modules sounds like a powerful new method to solve some of the separation of concern problems currently present in most development projects. It takes the best of Subject-Oriented Programming and makes it usable. It builds on top of Object-Oriented programming and is thus optional. Since it is in its early stages it is hard to tell if it will be over-used and thus complicate the code against the intention. There might also be some problems with class inheritance; what if I want/need to inherit from a class after it has been combined with a hyper slice? Also some research needs to be done in how to easily compose hyper slices. Hyper modules helps dealing with functional concerns, but not necessarily with non-functional concerns as those dealt with in Aspect-Oriented Programming.

5 Superimposition

As stated earlier, the main objective of software engineering is to improve software quality, reduce costs, facilitate easy maintenance and evolution of the software product. One way of doing this it to develop re-usable components. The components are often used in many projects and thus they are often very complete and well tested, which helps insuring low costs and improved software quality. Unfortunately, components are not the holy grail, as they often have to be adapted to the system requirements - which often requires understanding of how the components are implemented.

In the paper that we have examined, five criteria are used for classifying various ways of adapting components. These are:

- **Transparency**; the adaptation between the component and the user should be as transparent as possible, thus the components should not feel alien to the product.
- **Black-box**; the user shouldn't need to know about the internal structure of the component, only its interface.
- **Composability**; the adaption technique should be composable with the component without changes to the component; the composed component should have the same composability as the original; it should be possible to compose the adaptions.
- **Configurability**; it should be possible to configure the adaptation technique to fit the users needs.
- **Re-usability**; the adapted components should be re-usable, which is often not the case since the configuration is often intertwined in the generic adaptation.

There are three often used methods for component adaption; copy-paste, inheritance and wrapping, all with there advantages and disadvantages. With copy-pasting the component into the project or inheriting from the component, only the *transparency* requirement is fulfilled. Wrapping the object, on the other hand, doesn't help *transparency*, but somewhat fulfills the *black-box*, *composibility* and *re-usability* requirements. The reasoning behind this can be found in [Bosch, 1998], though it should be quite obvious if you have worked with any of these techniques.

5.1 The idea

Superimposition is a way of describing an adaptation as some kind of mapping. This makes it quite easy to compose different adaptations and satisfy the requirements stated above. Each object is described as an interface, a set of methods, a set of states (instance variables) and a mapping from the interface to the methods. The adaptation is performed by modifying these by standard mathematical operators. Instead of going into details with these, we recommend the interested reader to read the actual paper.

As simple example to demonstrate the idea, is a restriction of an interface: A restriction can be described as a set of interface methods that we want to keep, plus a function that actually performs this operation.

Similar rules can be made for other types of adaptations. We mentioned that these adaptations can be categorized in three different types: *Changes to component interface* (like function renaming), *component composition* (like delegation of request), and *component monitoring* (reacting on certain conditions in the state of a component).

5.2 Our Conclusion

Superimposition sounds like a good way to improve component adaptation and it also makes it possible to reduce the overhead when composing adapted components. This is because the binding can be done directly (only one wrapping) instead of a wrapping of a wrapping. It still requires, though, that you fully understand the interface of the component before you do the adaptation. If the components doesn't export enough in its interface you will still have to deal with the internals of the component, and if the component exports too much, it will be hard understanding the interface. Whether the method works well in practice is hard to say, but it sound promising.

When reusing components in a project one often has to write a wrapper to make the components compatible with the components already in the project. Sometimes this is also necessary to make data types compatible (like different representations for strings). We do not see how superimposition can solve this often-occurring problem.

6 Further reading

If there is interest to read more about the suggested methods, we suggest reading the actual papers, as well as some of the papers referred from these. We suggest reading the papers in the same order as we have dealt with them. This way you will gain the background for the later papers and you will also get a good idea how the theories are progressing.

7 Conclusion

After reading the four proposals on improving software development techniques, as described in this text, we come to the following conclusion: While all texts provide interesting new ideas, the stage in which the development of the ideas are, differ a lot. The proposed ideas are not all usable in their current form. Subject-oriented programming, for instance, is a nice idea, but it seems virtually impossible to make it usable in the near future. Hyper modules, seen as a weak version of Subject-Oriented programming, on the other hand seems like a technique that could be introduced in programming languages in a foreseeable future. Superimposition and Aspect-Oriented programming seem like techniques that are created as solutions for practical problems that the authors experienced. Both techniques could be useful and could be implementing in some application-specific tools. After some experience is gained with these techniques, we think they are good candidates for being introduced into current programming languages.

8 Acknowledgements

We want to thank our teachers Rein Smedinga and Jan Terlouw for their course and for further reviews of this paper.

References

Bosch, J. (1998). Superimposition: A component adaptation technique.

Harrison..., W. (1999). N degrees of separation: Multi-dimensional separation of concerns.

Harrison, W. and Ossher, H. (1993). Subject-oriented programming (a critique of pure objects). *OOPSLA*, 411–428.

Kiczales..., G. (June 1997). Aspect-oriented programming. *Proc. European conference on Object-Oriented Programming (ECOOP), Finland, Springer Verlag*, **LNCS 1241**.

Image Segmentation: Problems, Techniques and Evaluation Criteria

Timo Laman and Martijn Bodewes
{timo|bodewes}@fmf.nl

Abstract
A little over ten years ago [6] started a discussion that addressed severe problems within the field of Computer Vision. The main problem stated was how the research in the field should continue. Image segmentation techniques and how to use them correctly were named as a key subject which has been neglected. This paper discusses the problem stated, focusing on the field of image segmentation. It also gives a short overview of the state nowadays. A few papers on segmentation techniques and their evaluation are discussed and their relevance to the problem stated is determined.

1 Introduction

1.1 Problems in computer vision

A little over ten years ago, [6] signalled a lapse in the scientific development of the Computer Vision research field. The authors felt that Computer Vision, though potentially a powerful tool with a great many possible applications in a variety of fields, failed to become a mature science. The problems are attributed to the preoccupation of researchers with theory, neglecting the experimental aspects.

According to the authors, the entire research field suffers from three general problems:

ignorance in many algorithms, no attempt is made to use higher level knowledge of the scene or represent this knowledge in a way that is useful

myopia often, algorithms rely on properties that hold only in specific areas of the image, and fail to look at the "big picture". The authors call this *spatial* myopia (or short-sightedness). Another problem specific to computer vision involving motion is *temporal* myopia, the failure to make full use of all data in the sequence of input frames

naiveté the inclination of some researchers is to accept statements about the usefulness of algorithms without proper experimental justification

In the ensuing dialogue about the future of computer vision as a science, other researchers in the field gave their opinion. [10] focuses on the "naiveté" problem, saying that in order to become a mature science, rigorous experimental methodologies need to be developed based on error analysis and that in order to achieve this, the tendency to develop algorithms with a large number of parameters needs to be stopped.

[1] and [5] disagree with [6], stating that in their opinion computer vision is continuing to grow into a mature science and that the problems signalled by [6] are inevitable in a science that is relatively young. This vision is shared more or less by [3].

Finally, [7] states that the problems might also be caused by computer vision being a rather "closed" area of research, in which fresh ideas get little chance to blossom because they are simply never picked up by other researchers.

1.2 Relation to image segmentation

One of the subtopics of the computer vision research area which the [6] mention as a neglected problem is image segmentation. The stated problems of ignorance, myopia and naiveté are all applicable to this area in some way.

This paper will explain why image segmentation is important for almost all applications of computer vision systems. Also, it will show how the problems [6] mentions apply to this particular research field. Finally, it will try to give on overview of possible improvements and research conducted in the area over time.

2 Image segmentation in Computer Vision applications

In order for a computer program to derive useful information from an image, it is important that the image be divided into the objects that are of interest to the particular application, and parts which are of no consequence and can therefore be ruled out from the rest of the processing. For example, to count the number of coffee beans in a picture of coffee beans, the image has to be divided into the beans (the objects of interest) and the background (the surface on which the beans lie). The subsequent processing would consist of counting the separate (unconnected) objects in the image.

The importance of good segmentation results is evident. Even in this simple example, a bad segmentation algorithm would lead to incorrect results. If, for example, two coffee beans lying close to each other – or even on top of each other in a pile – would be considered as one object, the result of the count would come out too low. This is clearly visible in figure 2: the number of different regions is lower than the number of beans, simply because beans close to each other cannot be distinguished.

Image segmentation thus involves classifying pixels in the input image according to in which object/region they are located.

One of the difficulties in image segmentation is that a unique segmentation usually does not exist. Pixels in the input image can be grouped in different configurations depending on the application, and the objects that are to be expected. This means that no "general" image segmentation tech-

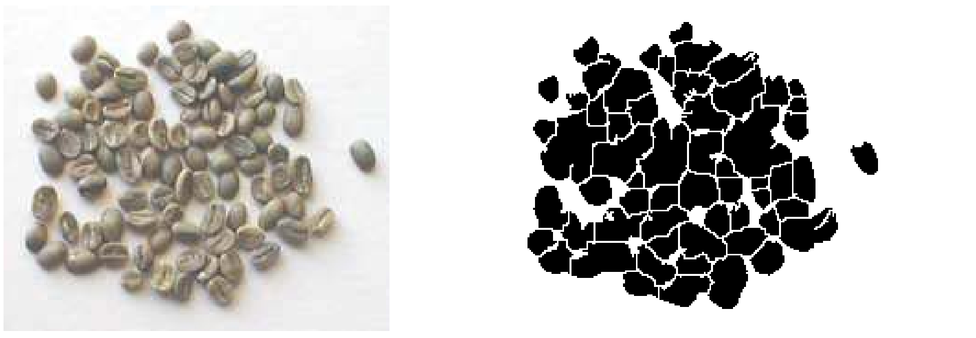


Figure 1: An image of coffee beans segmented using thresholding and watersheds

nique can be constructed, and that the segmentation method used has to be tuned to the specific application. In this respect, [1] disagrees with [6], stating that effort spent on coming up with a general way to solve the problem is wasted and that concentrating on segmentation for a particular purpose would be more fruitful; according to this article, general image segmentation is too ill-defined a problem to find a solution for.

3 Identified problems in image segmentation research

In [6], image segmentation is recognized as one of the most important issues of computer vision. The main concerns expressed in the dialogue are the following:

- A lot of research has been done that assumes either that a good image segmentation technique is available – which is, unfortunately, rarely the case – or that image segmentation is not a necessary at all – which results in solutions working only on images with no background, which are rare. Therefore, this research has no immediate practical value, and can't be tested on real-world example images. Instead, synthesized and unrealistic test input images have to be used.

- There is a lack of objective and automated evaluation criteria for image segmentation. It is hard to tell if a particular segmentation is "good". This is a result of the lack of experimentation in the research, which makes it unnecessary to define evaluation measures, and thus to think about quality criteria for the segmentation technique.

- Image segmentation techniques frequently rely on local intensity properties in the image, like sudden contrast differences (in the case of edge detection) or connected regions of approximately the same intensity value; complex objects however might contain a large range of intensities due to lighting or texturing of the scene.

- Researchers are too preoccupied with low-level filters and operators, and fail to incorporate higher level knowledge of the application domain in the segmentation phase of the process.

The first problem is not a problem of image segmentation per se, but it affects algorithms that rely on an image being segmented correctly. It is highly related to the second problem: the lack of evaluation criteria for segmentation techniques prevents experimenting with the techniques, which makes it hard to test higher level vision algorithms on real images (because no robust and tested image segmentation techniques are available). Thus, the naiveté in image segmentation area causes naiveté in other vision areas of research as well.

The last two problems, which might be considered forms of myopia and ignorance respectively, are also related to each other. Whereas in the image itself no information is contained other than the intensities at different pixels in the image, higher level knowledge might help to identify objects in the image with differences in intensity if that particular object is expected to have these differences and is likely to be in the image.

4 Research in image segmentation

In this section, an overview of research into some aspects of image segmentation is given.

4.1 Segmentation techniques

A lot of different criteria and approaches can be used to achieve image segmentation. Pixels can be assigned to a region because of a common property in intensity: examples of this *region-based* approach are thresholding, where simply all pixels with an intensity between to threshold values are considered to be the "foreground", or watershed segmentation, where regions are "grown" from a certain point within the region and separated at the places where the borders of the region touch.

An *edge-based* approach can also be used: in this case, regions are defined by enclosing boundaries composed of pixels with a particular property.

Finally, a *boundary-based* approach, where the boundaries between two regions are located by some transition of properties of the pixels at that boundary, can be used to identify regions.

The above methods can be applied to achieve an initial segmentation of the image. In [9] a framework for image segmentation is given which can refine such an initial segmentation using rules based on the knowledge about the application domain, for example in order to merge regions which actually belong to the same object.

4.2 Segmentation evaluation

Of the problems described in the previous section, the problem about a lack of evaluation criteria is probably the most interesting. Defining a set of evaluation criteria for segmentation techniques allows for experimenting on images and fine-tuning algorithms to specific application areas. Also, evaluation criteria can serve as a guideline to follow when designing new segmentation algorithms.

Empirical evaluation strategies can be divided in two classes ([4]): "goodness" and "discrepancy". The first strategy involves a measure that can be taken from the image directly, like for example the deviation of contrast inside each region in the segmentation. This way one is able to determine whether all pixels in the region really have a common property. For this strategy, no a priori knowledge of the image is necessary, so it can be used within a working application to actively monitor the correctness of segmentations.

The "discrepancy" strategy on the other hand does require a priori knowledge. In this strategy, the discrepancy between a precalculated segmentation (made, for example, manually by an expert in the particular application area), and the output of the algorithm run on the same image is determined. The less the discrepancy, the better the segmentation algorithm performs.

4.2.1 Goodness measures

Goodness measures can be calculated without having a reference segmentation. Because a lot of different criteria for image properties can be used to segment an image, different measures will have to be defined for different kinds of images or applications.

An example of a measure that is defined in [8] is the Inter-region contrast measure I. In this criterium, the total contrast of all regions in the segmentation is divided by the total area of the image, as follows:

$$I = \frac{\sum_{R_i} A_i c_i}{\sum_{R_i} A_i}$$

where

$$c_i = \sum_{R_j} \frac{l_{ij}}{l_i} \frac{|m_i - m_j|}{m_i + m_j}$$

In these equations, the R_i are the different regions in the segmentation, A_i are the areas of the regions, m_i the mean intensity of the pixels in region R_i, l_i the perimeter of R_i and l_{ij} the length of the border between regions R_i and R_j.

I increases when the contrasts of neighbouring regions increase, and because it is normalized, it takes a value between zero and one inclusive. Thus, a higher contrast between neighbouring regions is considered better, and low contrast might mean that the boundary between two regions should be moved or that two regions should be split or merged.

In [8] a few different criteria are evaluated on the segmentation of a number of different images, and the performance is measured by comparing the goodness measure to the goodness measure computed for a reference segmentation.

The example above is just one of the measures described. The fact that different measures have to be used for different kinds of images can be seen as follows: suppose you create a segmentation of a textured image, consisting of two adjacent checkerboard patterns, one consisting of 2x2 pixel squares and the other of 4x4 pixel squares. The expected segmentation then would identify the two patterns as different regions. However, since the mean intensity of both patterns is the same, the Interregion measure would characterize this ideal segmentation as a bad one.

4.2.2 Discrepancy measures

Empirical discrepancy measures were around as early as 1982 ([9]) and are used for example in [11].

The measure described in [9] compares the output of a particular segmentation algorithm, consisting of regions $\tau_i, 1 \leq i \leq M$, to a reference segmentation consisting of regions $R_j, 1 \leq j \leq N$. Let $A(r)$ denote the area of a region r. An *under-merging error* U can be calculated as

$$U = \sum_{j=1}^{M} \frac{(A(R_k) - A(\tau_j \cap R_k))A(\tau_j \cap R_k)}{A(R_k)}$$

Here, $a \cap b$ denotes the overlap of (the part of the image shared by) regions a and b.

R_k in the above formula is defined, for every τ_j, to be the region in the reference segmentation which fits best to the region τ_j, i.e.,

$$A(\tau_j \cap R_k) = \mathrm{Max}_{1 \leq i \leq N} A(\tau_j \cap R_i)$$

The under-merging error thus increases with the amount of area contained regions that do not correspond to regions in the reference segmentations.

Similarly, an *over-merging* error O is defined as

$$O = \sum_{j=1}^{M} A(\tau_j) - A(\tau_j \cap R_k)$$

This increases with the amount of area contained in regions that overlap multiple regions in the reference segmentation.

Both measures have zero as a lower bound (consider, for example, a segmentation which corresponds exactly to the reference segmentation: then $A(R_k) = A(\tau_j \cap R_k)$ for every j), and an upper bound of the area of the entire image, A_i.

A composite error value can be determined by combining the normalized values of the error measures, for example by taking the value $1/A_i\sqrt{O^2 + U^2}$.

4.2.3 Combinations

In [4], an approach is described combining different empirical strategies with other, analytical, measures like running time, memory requirements and analyzed behaviour of algorithms to specific input types to yield a measure that can be used to find an acceptable trade-off between different quality measures. To this end a function H is used to compute a weighted average of various evaluation criteria:

$$H(a_{\vec{p}}, I) = \Phi(f_1(a_{\vec{p}}, I), \ldots, f_n(a_{\vec{p}}, I))$$

Here Φ combines the output of some evaluation functions f_i into a single value. $a_{\vec{p}}$ is an image segmentation algorithm a with a list of parameters $\vec{p}$, and I is a set of reference segmentations (i.e., ground truths) to be used for discrepancy measures.

4.3 Use of higher level knowledge in segmentation techniques

A subject which still seems not very well researched is the use of higher level knowledge of the image in segmenting it. Usually, some higher level knowledge is inherently used (for example, when segmenting an image using a fixed intensity threshold).

In [9] a technique is described to refine a segmentation using a system of rules that can act on different properties of regions, which are calculated in an earlier step. By acting on specific conditions in the segmented data, regions can be merged or split. Higher level knowledge can be used to control the way regions are merged or split. The technique essentially consists of a framework to let the merge or split actions be governed by knowledge about what kind of regions might be available in the input image.

This way of refining the segmentation can be used to propagate knowledge obtained in a higher level of the algorithm to the segmentation phase. However, since it is a framework rather than a segmentation technique in itself, it needs quite a lot of additional work to incorporate it in a vision system satisfactorily.

In [2], use of a priori high level knowledge of a particular application domain is demonstrated in combination with active contour image segmentation, an edge-based segmentation technique that is based on fitting a curve around the region of interest by using some kind of energy minimizing function.

The technique is used on medical data obtained by CT, MRI or PET scans. In these kinds of images, different objects are frequently hard to identify automatically because often there is little variation of intensity between different objects.

Active contour segmentation relies on certain parameters for the fitting function. In semi-automatic systems, these parameters have to be chosen by a (human) expert monitoring the system. The system proposed in [2] automatically configures these parameters based on a semantic network containing anatomical information, such as size, intensity and relative location of organs or parts of them. A crude initial segmentation is used to choose a candidate for each region, after which active contours are used to reach the final segmentation.

This example again shows that knowledge-based segmentation is very specific to particular application domains, and that a general segmentation system is not readily devised.

5 Conclusion

In the past twenty years a lot of research has been done in the field of image segmentation. New segmentation techniques are being developed in great numbers. The claim in [6] that there are no evaluation criteria to determine if a technique is good enough seems to be obsolete; [9] already proposed a measure that is useable and as seen in [4] and [8], the research in this field is far from inactive. Naiveté is therefore not such a problem anymore, at least, from the perspective of image segmentation.

The existence of these measures allow for experiments with segmentation algorithms to find better techniques, and to tune existing techniques to specific applications. Of course, data to experiment on is also needed, as well as willingness from researchers to engage in experiments.

The problem of myopia is not quite as outspoken in image segmentation as perhaps in other fields of computer vision; it is difficult to be able to say anything about global properties of an image at such an early stage. Perhaps it should just be accepted as in [1] and [5] that the available data is usually myopic and that relying on local properties in images is inevitable. Also, the use of high-level knowledge may help in this respect to relate regions to each other.

On the problem of ignorance, and the use of higher level knowledge in vision and image segmentation in particular, not much information is available. A lot more research can and should probably be done in this area. However, it should be kept in mind here that higher level knowledge usually is very specific to the application, and therefore it is difficult if not impossible to devise a segmentation algorithm using high level knowledge that can be generally used.

References

[1] Y. Aloimonos and A. Rosenfeld. A response to “ignorance, myopia, and naivete in computer vision systems” by r. c. jain and t. o. binford. *CVGIP*, 53(1):120–124, January 1991.

[2] Riccardo Boscolo, Matthew S. Brown, and Michael F. McNitt-Gray. Medical Image Segmentation with Knowledge-guided Robust Active Contours. *Radiographics*, 22(2):437–448, 2002.

[3] K.W. Bowyer and J.P. Jones. Revolutions and experimental computer vision. *CVGIP*, 53(1):127–128, January 1991.

[4] Mark Everingham, Henk Muller, and Barry Thomas. Evaluating image segmentation algorithms using monotonic hulls in fitness/cost space. In Tim Cootes and Chris Taylor, editors, *Proceedings of the 12th British Machine Vision Conference (BMVC2001)*, pages 363–372. BMVA, 2001.

[5] T.S. Huang. Computer vision needs more experiments and applications. *CVGIP*, 53(1):125–126, January 1991.

[6] R.C. Jain and T.O. Binford. Dialogue: Ignorance, myopia, and naivete in computer vision systems. *CVGIP*, 53(1):112–117, January 1991.

[7] M. Kunt. Comments on dialogue, a series of articles generated by the paper entitled ”ignorance, myopia, and naivete in computer vision”. *CVGIP*, 54(3):428–429, November 1991.

[8] Hélène Laurent, Sébastien Chabrier, Christophe Rosenberger, Bruno Emile, and Pierre Marché. Etude comparative de critères d’évaluation de la segmentation. 2003.

[9] M.D. Levine and A.M. Nazif. An experimental rule-based system for testing low level segmentation strategies. *Multicomputers and Image Processing*

Algorithms and Programs, pages 149–160, 1982.

[10] M.A. Snyder. A commentary on the paper by jain and binford. *CVGIP*, 53(1):118–119, January 1991.

[11] M.H.F. Wilkinson. *Digital Image Analysis of Microbes*, chapter Automated and Manual Segmentation Techniques in Image Analysis of Microbes. John Wiley & Sons, 1998.

Using Force-Directed Methods For Drawing Graphs

Michiel Koning and Maarten Everts

Department of Computer Science
Rijksuniversiteit Groningen
Blauwborgje 3
9747 AC Groningen
{m.g.koning, m.h.everts}@student.rug.nl

Abstract
For many problems in information visualization, a well laid out graph can provide insight into the data. This paper discusses one family of algorithms to find an aesthetically pleasing drawing for a graph: force-directed layout algorithms. The basic force-directed algorithm is fairly slow and some algorithms which improve this performance are presented. The paper is concluded with an overview of which algorithm to use for which type of graph.

Keywords: Graph drawing, layout algorithms, force-directed layout algorithms, performance improvement

1 Introduction

In the field of information visualization, there are many types of data that can be represented as graphs. In order to obtain insight into this data, the elements of the graph must be properly positioned in a two- (or three-) dimensional space. There are many algorithms available that try to achieve this goal, by, for example:

- minimizing edge crossing
- minimizing edge lengths
- minimizing link bends
- maximizing symmetries

The aesthetic properties they try to achieve can sometimes be contradictory.

There are several types of graph layout algorithms, all having their own merits and application areas. For example, directed graphs are very suitable to be laid out by hierarchical layout algorithms. The most widely used hierarchical algorithms are based on [Sugiyama and Tagawa, 1981]. For an overview of existing algorithms for graph drawing, see [Battista et al., 1999].

In this paper, we will focus on one type of graph layout algorithms: force-directed methods. We will first discuss the simplest type of force-directed layout, and will then discuss several enhancements to the basic algorithm that improve upon it in different ways, mostly to speed up the algorithm.

2 Force-directed layout

Force-directed layout algorithms are a class of layout algorithms that try to obtain an aesthetically pleasing layout by representing the vertices of a graph as physical objects subject to various forces. The forces used differ from implementation to implementation. All have their particular aesthetic properties.

In an iterative manner the vertices are moved toward the direction the sum of forces exerted on them until the system reaches a stable configuration. A stable configuration often shows symmetries in the graph even though the algorithm does not specifically search for symmetries. This is one of the reasons why force-directed layouts are quite popular. Another reason is that the basic idea is simple and thus easy to implement.

2.1 Basic spring-embedder

The simplest force-directed algorithm uses a combination of electrical and spring forces. This is called a spring-embedder model, and was first introduced by Eades in [Eades, 1984]. Nodes are considered to have mutually repulsive charges and edges are modeled as springs that attract connected nodes.

Say $\Delta(v,w)$ is the distance vector between two nodes v and w and $\|\Delta(v,w)\|$ the Euclidean distance. The repulsive (electrical) forces between each pair of nodes are inversely proportional to the distance, so the force vector is:

$$F_{elec}(v,w) = -\lambda_{elec} \frac{\Delta(v,w)}{\|\Delta(v,w)\|^2}$$

Between nodes connected by edge (v,w), there is an attractive (spring) force directly proportional to the difference between the distance and the zero-energy length of the spring (Hooke's law):

$$F_{spring}(v,w) = \lambda_{spring} \frac{\Delta(v,w)}{\|\Delta(v,w)\|} (\|\Delta(v,w)\| - l)$$

Here λ_{elec}, λ_{spring} and l are parameters: λ_{elec} denotes the strength of the electrical repulsion, λ_{spring} represents the stiffness of the spring and l is the natural (zero energy) length of the spring.

2.2 Magnetic-spring model

One advantage force-directed layout algorithms have over other algorithms, is that they work well with undirected graphs. They also work for directed graphs, but for directed graphs it is desirable that the edges point in uniform directions. Sugiyama and Misue [Sugiyama and Misue, 1994] proposed an extention of the basic model that tries to enforce this. In this extension the edges are, besides springs, modelled as magnetized needles and a magnetic field is present that acts on the needles. We will discuss this algorithm here, because it is an interesting modification and is a good example of using a different set of forces for a force-directed layout algorithm.

The magnetic force is orthogonal to the edge and depends on the angle α between the edge and the magnetic field (see fig. 1). The result is that the edges rotate to align with the magnetic field. The formula for the force is:

$$F_{mag} = \lambda_{mag} \alpha^c \|\Delta(v,w)\|^2 \perp(v,w)$$

Again, (v,w) is an edge between nodes v and w, $\Delta(v,w)$ is the distance vector between v and w and $\perp(v,w)$ denotes the unit vector orthogonal to (v,w). The parameters λ_{mag} and c allow to tune the force.

Figure 2 shows some of the possible magnetic fields. Each has different purpose, for example a concentric magnetic field can be used to emphasize cycles in a directed graph and a vertical parallel field results in in a hierarchical-like tree layout.

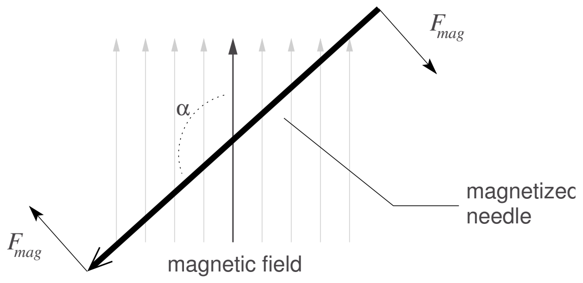


Figure 1: The force on a magnetized needle

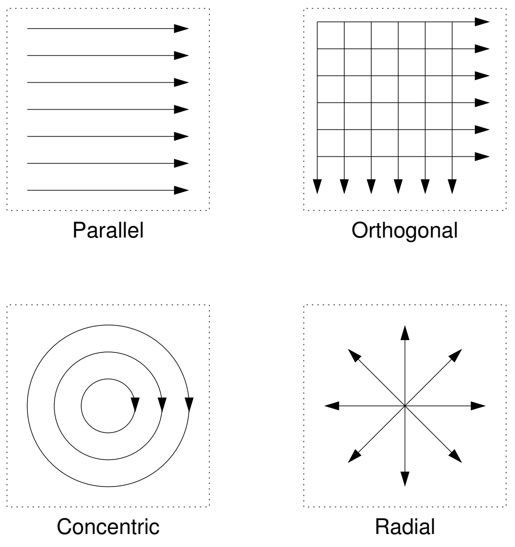


Figure 2: Types of magnetic fields

3 Fruchterman-Reingold

In [Fruchterman and Reingold, 1991], Fruchterman and Reingold discuss two enhancements of the basic force-directed layout algorithm. First, they introduce cooling (similar to, but different from simulated annealing[1]) and second, to speed up the spring-embedder algorithm, they introduce the grid-square algorithm.

3.1 Cooling

It is possible for the normal spring-embedder to get stuck in a local optimum, because of the fact that it always moves in the direction of the forces exerted on it. To make the spring-embedder able to get out of these local minima, a certain degree of randomness is needed when moving the vertices. The degree of randomness in movement should decrease over time, since the layout then steadily approaches the real optimum. The parameter which controls the randomness of the moment is called temperature, and the process of decreasing randomness is therefore called cooling. This technique is taken from simulated annealing, but was adapted to spring-embedders with respect to the direction of movement. Movement is random in simulated annealing, and movement which increases the total energy (energy is minimized) is rejected with a certain probability. Fruchterman and Reingold calculate the direction in which a vertex should go, but add a random displacement. The temperature they use only controls the maximum amount of displacement. The idea is that the layout slowly approaches the ideal one, and movement can be restricted during later iterations of the algorithm. This is faster than simulated annealing, because less time is spent on going in directions that do not improve the global energy total.

3.2 Grid-square

Another adaption Fruchterman and Reingold made to the spring-embedder, to speed the algorithm up, is the grid-square algorithm. In this algorithm, the total area for the graph is divided into squares with a certain width. When calculating the repulsive forces on a vertex v in a certain square, only vertices in adjacent squares are considered. For each vertex in the adjacent squares, their distance to p is calculated. Vertices only influence each other within a circle with a certain radius. When a vertex is outside the influence radius of v, it is too far away from v and the repulsive force it contributes to the movement of v is neglected.

The idea is illustrated in figure 3. v is the vertex in consideration. q, s and r are other vertices. Since r is not in one of the adjacent squares, it is not considered. q and s are considered; however, only q is within the influence radius. So, only q influences the movement of v.

This addition can significantly speed up the algorithm, since the complexity of the repulsive force-phase of the algorithm is no longer $O(n^2)$, where n is the total number of vertices.

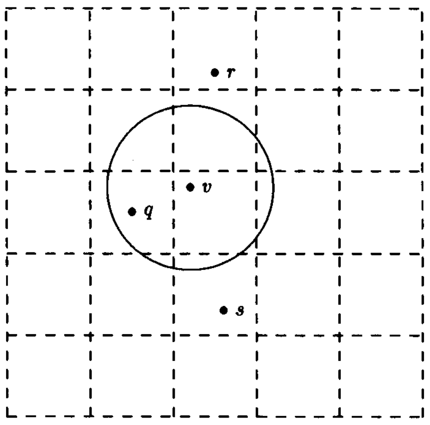


Figure 3: The grid-square algorithm

4 FADE

Using electrical repulsion in the model for the force-directed algorithm has the consequence that the complexity of one (!) iteration is $O(n^2)$, where n is the number of vertices (the position of every node has to be compared with each other node). As a result, simple force-directed layout algorithms are not very suitable for graphs with a large number of

[1] Optimisation technique introduced by Kirkpatrick in 1983 [Kirkpatrick et al., 1983] which applies statistical mechanics methods to find an approximate optimal solution to a problem. Typically a thermodynamic analogy is used for the model system under study and the task of finding an optimal solution is mapped to that of finding the energy-neutral state of the thermodynamic system.

vertices. In [Quigley and Eades, 2001] an algorithm called FADE is presented that aims to cure this.

The idea is to approximate the repulsive forces on a node by considering the influence of groups of other (more distant) nodes instead of every other node. To determine which nodes are close to the node under consideration and which nodes are more distant (and should be grouped together) the nodes are placed in a tree structure using recursive space decomposition.

4.1 The algorithm

To determine the electrical repulsion force on a node, the space decomposition tree is recursively visited. A treenode with children is called a pseudonode or supernode. The closeness of a pseudonode is determined by using a *tolerance criterion.* For a pseudonode we test $s/d \leq \theta$, where s is the width of the area for the pseudonode, d is the distance between the current node and the center of mass of the pseudonode and θ is the fixed-tolerance parameter. If $s/d \leq \theta$, then the internal nodes are ignored and the force contribution of the pseudonode is added to the cumulative force for that node. Otherwise, the pseudonode is resolved into its daughter pseudo-nodes (sub-trees), each of which is recursively examined.

Figure 4 shows an example. In this example a quad-tree space decomposition is used and the parameter θ has the value 1.0. Node 5 is compared with the pseudo-node p1. The weight of the pseudonode is the cumulative weight of the leaves in its sub-tree, in this case 4. The d_1 value is the distance between the node and the pseudonode. In this case $s/d = 0.80$. As this is smaller than the value 1.0 for θ the non-edge force between this node and the pseudo-node p1 are calculated. But for node 6, when comparing it with pseudonode p1, $s/d = 1.46$ and it does not fulfill the criterion $s/d \leq \theta$.

4.2 Results

In their paper, the authors claim that it is much faster than the basic spring-embedder and that they were able to use it on very large graphs (in the neighbourhood of 100000 nodes). However, in the results they present in the paper it is not clear whether building the quad-tree is included in the computation times. Building this tree will probably introduce some overhead, especially for small to medium sized graphs.

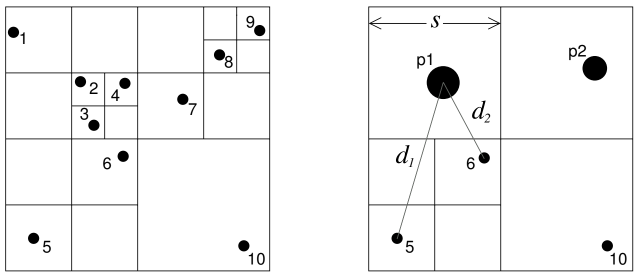


Figure 4: Space decomposition using quadtrees

5 Multi-scale approach

As mentioned earlier, a simple force-directed approach is unsuitable for drawing large graphs. The method proposed in [Harel and Koren, 2002] tries to address this by looking at the graph at different scales.

The idea is to iteratively update (beautify) the layout at different scales of the graph, starting at a coarse scale. This way both the global properties as the details of the layout are being considered. How to actually find these multi-scale representations is discussed next.

5.1 Finding a multi-scale representation

In a coarser representation of a graph $G = (V, E)$ the vertices that are drawn near each other in a nice layout should be grouped together. Add to this the observation that vertices that are closely related in the graph (i.e., the graph theoretic distance is small) should be drawn close together. So the problem of finding a multi-scale representation can be approximated by the *k-clustering* problem: partition V in k clusters so that the longest graph-theoretic distance betweeen two vertices in the same cluster is minimized.

Because it is useful for the algorithm to be able to identify clusters by a certain vertex in the cluster a solution for the *k-center* problem is used instead: choose k vertices of V, such that the longest distance from V to these k centers is minimized.

This problem is NP-hard[2], but fortunately there are heuristic solutions available.

We now have a method to find multi-scale representations of the graph. Note that a lower k will give a coarser representation of the graph. The next phase of the iteration is the beautification of the new representation.

5.2 Local beautification

For the local beautification step the Kamada and Kawai method is used. It is very appropriate because it relates every pair of vertices, so, when constructing a new coarse representation of the graph, it is not necessary to define which pairs of vertices are connected by an edge. This advantage has a price: $\Theta(|V|^2)$ memory is used, even when the graph is sparse.

As with all other force-directed-based algorithms, to find an aesthetically pleasing drawing, a certain energy function must be minimized. Since the objective of this layout algorithm is to draw nodes which are closely related near each other. The energy function should reflect this objective. Therefore, this function considers both Euclidean distance and the graph theoretic distance and tries to minimize energy for a neighbourhood of at most radius k (where k is determined by a constant). A vertex u is considered to be in the k-neighbourhood of a vertex v if the length of the path from v to u is at most k.

A (local) minimum for this energy is found if the derivative of the energy function is 0 with respect to both x and y:

$$\frac{\partial E_k}{\partial x_v} = \frac{\partial E_k}{\partial y_v} = 0, \quad \forall v \in V$$

To achieve this condition, the vertex with the highest sum of derivatives in both directions, and move this vertex towards the local minimum (similar to gradient-descent). This is done iteratively.

5.3 Algorithm overview

The algorithm by Harel and Koren for displaying large graphs can be summarized as follows:

[2] An optimization problem that relies upon the solution of an NP-complete problem. In that sense, NP-hard problems are at least as hard as NP-complete problems.

1. Compute All-Pairs Shortest Path (needed for both finding multi-scale representations and local beautification)
2. Setup a random layout
3. Define the coarsest level representation of the graph and do iteratively:
   (a) Determine vertices of current level representation
   (b) Perform local beautification
   (c) For each vertex, find the cluster to which it belongs and displace it randomly around the center vertex of this cluster
   (d) Define the next level representation for which local beautification will be performed

5.4 Results

With this algorithm, the authors were able to visualize very large graphs, within the neighbourhood of 1600 vertices and 2133 edges within 2 seconds on a Pentium III 1 GHz PC. They were also able to visualize much larger graphs (15606 vertices and 45878 edges), but this took slightly less than 5 minutes on the same system. The execution time also depended on the size of the k-neighbourhoods: the larger those neighbourhoods, the longer the execution time.

In figures 6 and 5, some results are displayed with different sizes for the k-neighbourhoods.

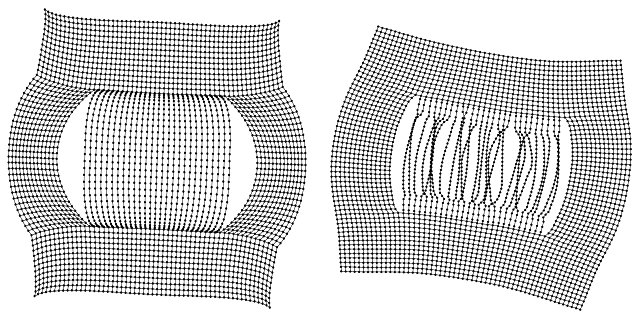


Figure 5: Multi-scale method applied to partial grid with (left) larger k-neighbourhoods than normal and (right) normal size neighbourhoods

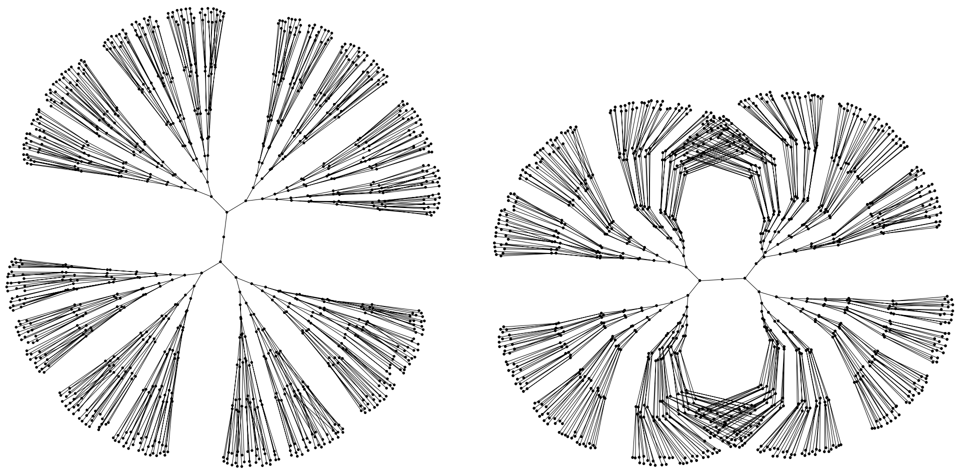


Figure 6: Multi-scale method applied to full binary trees with (left) entire graph as k-neighbourhood and (right) limited size neighbourhood

6 Conclusion and discussion

In this paper, we have first discussed the basic force-directed layout algorithm, followed by a discussion of several newer algorithms that try to improve performance over this basic algorithm. These algorithms were the Fruchterman-Reingold model, the FADE algorithm and the multi-scale method.

We cannot pick one of these algorithms to be applicable to all possible graph layout problems, because the choice of which force-directed layout algorithm to use largely depends on the properties of the graph one wants to display. These properties include size, edge-density and symmetry. For simple (smaller) graphs a normal force-directed layout algorithm will probably be suitable, mostly because it is so easy to implement. When the basic variant is not fast enough, one might consider the Fruchterman-Reingold model, but for very large graphs one would choose either FADE or the multi-scale method. Both claim to be much faster for very large graphs. However, comparing the theoretic complexity of these algorithms is difficult, since execution time depends largely on the chosen values of the parameters for these algorithms, such as the number of iterations or the coarseness. Also, quantifying the aesthetics of the results is very hard and very subjective.

Of course, comparing these algorithms based only on the contents of the papers in which they are presented is very difficult. It would be very interesting to compare the performance of these algorithms using real-world graph data (such as data sets from biology) with respect to execution time, aesthetic properties and, not to be underestimated, the amount of time spent on fine-tuning the parameters for these algorithms to get satisfactory results.

This paper only discussed determining layouts of graphs. However, displaying and exploring graphs is an entirely different field of research. See [Herman et al., 2000] for an overview of graph visualization and navigation techniques.

References

Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G. (1999). *Graph Drawing, Algorithms for the Visualization of Graphs*. Prentice Hall. ISBN 0-13-301615-3.

Eades, P. (1984). A heuristic for graph drawing. *Congressus Nutnerantiunt*, **42**, 149–160.

Fruchterman, T. M. J. and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software - Practice and Experience*, **21**(11), 1129–1164.

Harel, D. and Koren, Y. (2002). A fast multi-scale method for drawing large graphs. *Journal of graph algorithms and applications*, **6**(3), 179–202.

Herman, Melançon, G., and Marshall, M. S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, **6**(1), 24–43.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, **220, 4598**, 671–680.

Quigley, A. and Eades, P. (2001). Fade: Graph drawing, clustering and visual abstraction. In *Graph Drawing, 8th International Symposium, GD 2000, Colonial Williamsburg, VA, USA, September 20-23, 2000, Proceedings*, vol. 1984 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-41554-8.

Sugiyama, K. and Misue, K. (1994). Graph drawing by magnetic-spring model. Tech. Rep. ISIS-RR-94-14E, Institute for Social Information Science.

Sugiyama, K. and Tagawa, S. (1981). Methods for visual understanding of hierarchical systems. *IEEE Trans. Sys., Man, and Cybernetics*, **SMC 11**(2), 109–125.

FRIENDSHIP - FRIEND OR FOE: A RESEARCH INTO THE STRUCTURES OF FRIENDSHIP AND THEIR EFFECTS ON PRODUCTIVITY

RICK OOST
rickoost@gmail.com

NIELS HAGEMAN
nielshageman@gmail.com

This article will document our research on how to improve productivity through the use of friendship relations within R&D teams and advise managers in this matter. The first phase focussed on investigating the influence of friendship on productivity; the second part focussed on the attributes the management should take into consideration when structuring R&D teams to maximize the positive influence of friendship, thereby increasing productivity. The research was conducted using data from extensive questionnaires, filled in by about 200 members of R&D teams.

INTRODUCTION

It is always the goal of companies with an R&D department to improve the efficiency of their project teams because this is critical to keep up with business competitors. It is shown that of the contemporary medium to large sized companies, over eighty percent use a team-based approach. This percentage is even higher in companies where the focus lies on R&D (Cohen and Bailey, 1997). During the life cycle of a team, certain structures of friendship relations will form and evolve, even if the members of the team do not know each other beforehand. Most managers try to encourage friendship, because they feel this has a positive effect on communication and on productivity (Berman, West, and Richter Jr, 2002). Our research into this matter has had two goals. The first was to investigate if there was a relation between various structures of friendship relations and productivity; the second was to establish how these could be manipulated to make their effect on productivity as positive as possible.
There has been previous research into this matter, but their results have proven contradictory. Some research shows that friendship has an adverse effect on the productivity of a team, because the focus changes to social interaction, instead of the team's task. Other research shows that friendship has a positive effect (Kratzer, Leenders, and van Engelen, 2004), for example because the team members are more committed and more cooperative, which increases performance (Jehn and Shah, 1997), or because cooperation between friends results in tasks being completed more quickly (Shirase, Nagafune, Wakamatsu, and Arai, 2000).

OBJECTIVE

The objective of this study has been to formulate an advice to the management of the organizations participating in this study on how to use friendship bonds within R&D teams to improve their productivity. In order to achieve this objective, we have used the results of extensive questionnaires filled out by the R&D employees of the participating companies.
We started by analyzing the problem and finding the relevant factors and variables.

- There is insufficient productivity.

The authors would like to thank Jan Kratzer for his assistance with our research, research report and this article. We would also like to thank John Kizito and Jens Rasmussen for their feedback on this article.

This problem description is rather abstract and broad and is not citing a possible solution.
Like any relationship, friendship can be modeled using a graph and has certain properties.

- Friendship
  - Structures
    Cohesion, Centralization, Segmentation

We have chosen the above three properties because of their use in prior research (Kratzer et al., 2004). The first step in our research was to see if these properties exerted an influence over the team productivity. We will now give a brief explanation of each of the properties.

- The cohesion of friendship bonds within an R&D team refers to the number of actual bonds in comparison with the total number of possible friendships.

- The degree of centralization is an indicator of to what degree a network is revolving around a single node. An example of a highly centralized network would be a situation where the only friendship relations are between a single person and all other persons (a “star” network with one person central and no relationships between the others).

- Segmentation refers to the formation of groups within a team that share close friendship bonds internally, but much less so to the “outside world”. Such a subgroup is also known as a clique.

Having finished the first step, we shifted our attention to the factors that influenced the formation of friendships. In this second stage, we looked for correlations between various properties such as age and both the number of friendship bonds in the team (The friendship score) and the average strength of these bonds. From all the attributes available for investigation, we decided to focus on the ones that described personal properties such as age. Some of these properties had to be dropped. Gender could not be used because there were only a few women among the nearly 200 participants. This made it impossible to draw statistically significant conclusions. Properties like specialization could not be considered because their values were not on an ordinal scale. After this process of selection and elimination, we were left with six suitable ones: Age, Degree, Fraction on team, Members, Number of teams and Time on team. These properties will be discussed in further detail in the section about the methodology used.

RESEARCH QUESTIONS

For this research, we posed two primary questions and one secondary question. The first primary question, corresponding to the first stage of the research was:

> *“What is the influence of cohesion, centralization and segmentation of friendship relations on the productivity of R&D teams?”*

The second one, corresponding to the second stage was:

> *“What is the influence of age, degree, fraction on team, members, number of teams and time on team on friendship relations?”*

These questions were broken up into smaller partial questions, each focussing on a certain aspect, to make them easily answerable. The hypotheses were based on these questions and are discussed further below. The secondary question we posed was:

> *“How can the effect of friendship relations between members of the R&D staff be used optimally in the structuring or restructuring of R&D teams to improve their productivity?”*

The answers to the primary questions will be the basis for the answer to this question, that will form the advice to managers.

HYPOTHESES

First primary question

Cohesion. As described above, cohesion is the ratio between the actual number of friendships and the number of friendships that are theoretically possible. Research indicates that information flows more freely between people who are friends (Zaccaro and Lowe, 1986). This is because of more contact and greater trust between them (Roloff, 1987; Danowski, 1980; Rawlins, 1983). We expect that if the ratio increases, the ability of the members of the team to work together will increase as well. We do not expect this increase to be linear, but instead level off and possibly decrease a little at very high levels of cohesion, because the focus of the team will be shifting to social interaction instead of the team's task (Jehn and Shah, 1997).

> *Hypothesis 1:* The larger the cohesion in a team, the higher the productivity.

Centralization. Since a high degree of centralization means that there are one or a few nodes that play a central role, we expect this to have a detrimental effect on efficiency for a few reasons. One is that it is possible that communication will run through such a central point, instead of immediately to the destination. Another reason is that when the central player temporarily or permanently vanishes, for example due to illness or reassignment, cooperation and communication structures within the team may fall apart.

> *Hypothesis 2:* A high degree of centralization will lead to a decrease in productivity.

Segmentation. Subgroups within a team tend to develop very efficient communication structures (Dearborn and Simon, 1958; Wilensky, 1967), however, communication with the members outside the group degrade usually (Kratzer et al., 2004). In some ways this can increase productivity, in other ways it can decrease, depending on the type of task these teams are working on. If the task is well partitionable for example, there may not be a negative effect or even a small positive effect, but if the task requires the whole team or large (clique-spanning) portions thereof to work together, there may be a negative effect because the communication between segments is inherently less then the communication within. If there is a high specialization within a team this view is supported (Kratzer et al., 2004). Unfortunately we do not have data on the type of work.

> *Hypothesis 3:* The degree of segmentation will not have a clear positive or negative effect on productivity. Instead, the effects will vary from team to team.

Second primary question

Age difference. We expect that if the difference in age is lower, more and stronger friendships will form. This because people of roughly the same age can be expected to have similar interests etc., giving a more fertile soil for a friendship to sprout in.

> *Hypothesis 4a:* The lower the age difference, the stronger the friendship bonds become.

> *Hypothesis 4b:* The lower the average age difference of all members, the higher the friendship score for the team.

Difference in education level. We do not expect the difference in age to make much difference in the formation of friendships. Despite having had different educations, people may still find common grounds between them.

> *Hypothesis 5a:* The difference in education level will have no noticeable effect on the strength of the friendship bonds.

Hypothesis 5b: The average difference in education level will have no influence on the friendship score of the team.

Fraction on team score. We expect the strength and number of friendships to increase if people spend a greater portion of their time on a team they both work in. More frequent contact should allow for more opportunities for friendships to bloom.

Hypothesis 6a: The higher the fraction on team score, the stronger the bonds are.

Hypothesis 6b: The higher the average fraction on team score, the higher the friendship score of the team.

Number of members on a team. We expect that the smaller the number of members a team has, the more easily friendships will form and strengthen within them. A smaller team creates a more personable environment where more people know each other.

Hypothesis 7a: The fewer members the team has, the stronger the friendships bonds.

Hypothesis 7b: The fewer members the team has, the higher the friendship score.

Number of teams people work on. We expect that if the number of teams people work on increases, friendships will form less easily. If one works on more teams, on average one spends less time on each one of them. Although the number of people they are in contact with increases, the contacts themselves will become more fleeting.

Hypothesis 8a: The higher the number of teams, the lower the strength of the friendship bonds.

Hypothesis 8b: The higher the average number of teams, the lower the friendship score.

Time on team. We expect that if people spend longer on a team, the number and strength of the friendships will increase. A longer period of being exposed to each other increases the chances that a friendship will be formed.

Hypothesis 9a: The longer they have worked on the team, the closer the bonds.

Hypothesis 9b: The longer they have worked on the team on average, the higher the friendship score.

METHOD

As stated before, we will base our research on the data from the questionnaires filled out by the R&D employees. One of the questions asked was for each employee to specify whom of the other team members he considered to be his friend[1]. This yields an $n \times n$ matrix for a team consisting of n persons. When person A considers person B as one of his friends, the cell in row A, column B will have the value 1. If he does not consider him a friend, the value will be 0. All the values on the main diagonal (corresponding to a possible friendship of a person with his- or herself) are defined as zero (no friendship). An example of a possible friendship matrix of a five person team is this: (The matrix presented here does not represent any actual team, the data is purely fictitious)

	1	**2**	**3**	**4**	**5**
1	0	1	1	0	0
2	1	0	0	0	0
3	1	0	0	0	1
4	0	1	0	0	0
5	0	0	1	0	0

[1]Since each employee was asked individually, the friendship relation is not necessarily symmetric. If person A considers person B his friend, the inverse is not necessarily true.

Because this expresses the abstract friendship notation as numerical data, it clears the way for the use of statistical analysis on the data sets. It also allows us to give more formal definitions of the hitherto informally specified properties. We will first define the properties of the first primary question.

Properties

Cohesion. We are defining the cohesion of a group as the number of friendship ties in the group divided by the total number of possible friendship ties. For a group of size n, the latter number is $n(n-1)$. This gives a cohesion factor on the scale of 0 to 1, where 0 denotes a group with no friendship relations at all and 1 a group where everyone is a friend to everyone.

$$\text{cof} = \frac{\#\text{edges}}{n(n-1)}$$

Centralization. To define centrality, we will look at the data as a graph, with the different employees being represented as vertices and the friendship relations as directed edges. To define centrality we will be looking at the indegree of the vertices. The indegree of a vertex is the number of incoming edges at that vertex. This corresponds to the number of people that consider the person represented by said vertex as a friend. To calculate the indegree d_v of vertex v by taking the sum of the values in the column corresponding to v. We then calculate the average indegree of all vertices. The next step is to calculate the quadratic difference of the indegree of each vertex with the average. The final step of the calculation is to sum all of the values obtained in the previous step and divide that value by $n-1$, where n is the number of employees in the team. This gives the following formula for the centralization factor:

$$\overline{d} = \frac{\sum_{v=1}^{n} d_v}{n}$$

$$\text{cef} = \frac{\sum_{v=1}^{n} (d_v - \overline{d})^2}{n-1}$$

The higher the factor obtained by this formula, the higher the centralization in the team is.

Segmentation. For the definition of segmentation, we will again be looking at the data as a graph, just like with centralization. To measure the segmentation, we will count the number of cliques within the graph, where a clique is defined as a complete sub graph of at least three nodes and having maximum size. Complete means that every pair of distinct vertices is connected by an edge, maximum means that we will consider a complete sub graph with four nodes as 1 clique (we could also consider it as four cliques with three nodes). To obtain the degree of segmentation, we divide the number of cliques by the number of maximum possible cliques, which for a graph with n vertices is:

$$\binom{n}{3}$$

This gives as a formula for segmentation:

$$\text{sef} = \frac{\#\text{cliques}}{\binom{n}{3}}$$

The higher the factor obtained by this formula, the higher the segmentation.

The three properties above could be directly calculated from the existing datafile. Because of the nature of the datafile, the six properties of the second primary question could not be directly extracted. The original dataset contains one record for each employee, listing its attributes. Because the second primary question is friendship bond oriented instead of person oriented, we need the properties per friendship bond instead of per person. For this goal, we created a simple program to parse the original person oriented and create two friendship bond oriented ones. The first generated dataset contained a record for each potential friendship bond, the second one the same data, but then averaged per team[2]. Since a friendship bond is a relation between two persons, it has two sets of attributes connected to it (one from each person). The way to calculate the value of the attributes per the bond differed per attribute:

- Age: The absolute difference of the ages of both persons. This value is an indicator for the age difference in the bond.
- Degree: The absolute difference of the degrees of both persons. This value is an indicator of the difference in education levels.
- Fraction on team: This attribute indicates the time a person works on this project as a fraction of the total time he works on all projects. For each bond, this is the sum of the values for both persons. This value is an indicator how much both persons work together.
- Members: The number of members the team has. This attribute is the same for both parties, so no additional arithmetic necessary.
- Number of teams: The sum of the values for both persons. This value is an indicator on how many different teams both persons work,
- Time on team: This attribute indicates how long a person has been a member of the team. For each bond, this is the sum of the values for both persons. This value is another indicator how much time both parties spend together.

Another conversion that was carried out on the dataset was the symmetrization of the friendship matrix. This was necessary to deal with the problem that A could consider himself a friend of B, but not vice versa. The symmetrization means that a friendship bond was considered to exist if one or both of the participants considered the other as a friend. The downside of this is that information is destroyed. To mitigate this problem, each potential bond in the new dataset was assigned a weight. If both persons of the bond considered each other friends (i.e., a mutual friendship), the bond was assigned a weight of two. If the friendship was one way, the potential bond was assigned weight one and if neither participant considered the other his friend, the potential bond was given a weight of zero.
In both stages of the investigation, we have performed a quantitative analysis of the data, using a fixed design.

Descriptives
There were three dependent variables (Productivity[3] in the first phase and friendship strength and friendship score in the second) and nine independent variables (Cohesion, centralization and segmentation in the first phase, team members, education level, time on team, fraction on team, number of teams and age in the second phase) involved in our research. Table 1, Table 2 and Table 3 show the respective tables of descriptives.
In several cases, missing data was encountered. Records that missed values for attributes that were under consideration in our study were excluded from the research.

RESULTS

In order to determine if a correlation existed between variables, linear regressions were used of the independent variables against the dependents. For this, we defined $\alpha = 0.05$ as a threshold to compare with the significance of the regression. If the significance exceeded our threshold, we concluded that there was no linear correspondence.

Cohesion
Linear regression shows that the degree of cohesion of friendship relations in a team has a positive effect on the productivity. Since the significance

[2]Instead of calculating a "real" average of the values, the sum of the values was multiplied by 1,000 prior to being divided by the number of members in the team. This was necessary for increased resolution of the average, since the datasets consist solely of integral values.

[3]The productivity for a team was calculated by averaging the values of all the employees of the team.

Variable	Minimum	Maximum	Mean	Std. Dev.	N
Dependant variable					
Productivity	3.4	6.4	4.544	0.868	31
Friendship					
Cohesion	0	0.6	0.260	0.209	31
Centralization	0	1.3	0.560	0.420	31
Segmentation	0	0.3	0.070	0.093	31

Table 1: The variables and their properties for productivity

Variable	Minimum	Maximum	Mean	Std. Dev.	N
Dependant variable					
Friendship strength	0	2	0.377	0.674	494
Attributes					
Team members	5	10	6.949	1.881	494
Education level	0	3	0.545	0.681	494
Time on team	2	20	4.597	3.096	494
Fraction on team	2	8	5.692	1.880	494
Number of teams	2	30	4.472	4.544	494
Age	0	4	0.765	0.744	494

Table 2: The variables and their properties for friendship strength.

is also within the established boundaries, we will accept hypothesis 1 as valid.

Centralization
The value for the significance that was yielded by testing hypothesis 2 is far above our defined maximum, which leads to the conclusion that there is no linear correspondence between centralization and productivity. This therefore invalidates hypothesis 2. It is possible that a nonlinear regression would yield a better fitting formula, possibly one within the bounds of significance. However, several non-linear curve fittings were applied to the data set, none of them yielding an acceptable match.

Segmentation
Our analysis shows that there is a positive effect between segmentation and productivity. This leads us to reject hypothesis 3 as invalid since we postulated that no clear trend would be visible. This can be caused by a lot of specialization within the team (Kratzer et al., 2004), and the work is partitioned with the partitions assigned to the subgroups.

Age difference
Linear regression of friendship score and strength against this variable yielded significance values far exceeding our established alpha threshold. From this we conclude that age difference has no significant influence on the formation of friendships. We therefore reject both hypotheses pertaining to age that postulated a significant influence. The lack

Variable	Minimum	Maximum	Mean	Std. Dev.	N
Dependant variable					
Friendship score	5	10	6.030	1.571	33
Attributes					
Team members	0	1500	553	450	33
Education level	0	1666	592	403	33
Time on team	2000	11142	4295	2630	33
Fraction on team	2000	7333	5218	1877	33
Number of teams	2000	15600	5421	4518	33
Age	0	1800	803	420	33

Table 3: The variables and their properties for friendship score.

of a relation can possibly be attributed to the fact that in workplace friendships age does not matter as much as in traditional friendship (Simonetti and Ariss, 1999; Crampton and Mishra, 1999; Matheson, 1999).

Difference in education level
Our analysis of this factor showed that no linear correspondence can be assumed because of the high significance value. Both hypotheses corresponding to the difference in education level are therefore validated, since they already postulated the absence of a relation.

Fraction on team score
Our analysis here found the lowest possible significance values for both regressions. We can therefore safely assume that there is a linear relation between the percentage of time one spends working on a team and the formation of friendships therein. However, the coefficients of the relation were found to be negative: The lower the value, the higher scores. Since this is the diametric opposite of what we postulated in the hypotheses, they are rejected.

Number of members on a team
Our analysis of this factor again shows very low values for the significance with .000 and .009, indicating a plausible linear relation. The coefficients are sub zero, showing a negative line. This means that the more members a team have, the weaker and fewer the bonds become. The hypotheses are verified by these results, since this is exactly the relation we postulated. This therefore validates the hypotheses.

Number of teams people work on
Yet again a clear linear correlation was found here between the independents and the dependent. However, we see the same effect we saw with the "Fraction on team" independent variable. The relation is the diametric opposite of the one we postulated. We postulated that participating in more teams would lead to weaker bonds and a lower friendship score, but the opposite is true. We therefore reject the hypotheses. As with the fraction on team, we see the exact opposite of the expected effect. If we take that outcome into account, this outcome isn't surprising, considering the relation between both attributes.

Time on team
Our analysis here showed significance values exceeding the alpha threshold, although not as extreme as some of the other values for significance we encountered in these regressions. This means that no linear relationship exists between the time people worked on a team and the strength and number of friendships. Since the hypotheses postulated the existence of such a relation, we are forced to consider them invalid.

ADVICE

The first part of this research shows that to increase productivity, it is useful to put friends in the same team. This can be done in two ways, by influencing cohesion, or by influencing segmentation.
By putting as many friends as possible in the team the cohesion is maximized. Another approach to increase cohesion is to schedule activities aimed at the formation and tightening of friendship bonds within the team.
A manager can also select friends with the same kind of specialization and put them in one team, to maximize segmentation. This approach only works when the type of work is suitable for partitioning, as stated above.
Since the friendship structures within a team are not static but in a state of constant flux (generally the number of friendship bonds within a team will increase), it is also useful to periodically reevaluate and if necessary rearrange teams.
In order to aid in the formation of teams, the second part of our research has focussed on the properties of persons and teams that are likely to foster strong and numerous friendships. Out of the six attributes tested for influence, only three showed a clear linear relation with the strength and number of the friendship bonds. All attributes that had influence either influenced both (strength and count) or neither; no attribute influenced one but not the other. In addition, when there was a linear correspondence, the effect was the same on both dependents, i.e. there is not an attribute that positively influences bond strength, but negatively influences bond count or vice versa. This is a very good characteristic, because it means there will be no trade-offs between both.
The three characteristics that do not show a relation (being age, education level and the time spent on working on a team) can be safely left out of consideration when forming or changing project teams. The clear negative relation between group size and friendships means that whenever possible smaller groups should be preferred over larger ones. The research also clearly shows an improvement in friendships when team members are a member of multiple teams and do not spend all their time on a single team. This means that whenever possible workers should be assigned to multiple teams simultaneously. Where possible, both recommendations could be combined by splitting up a larger team into smaller ones and have part of the staff working on both teams.
Following these recommendations will likely lead to more and stronger friendships between employees and as a consequence, increased productivity.

FURTHER RESEARCH

Some interesting points have come up during this research project that may warrant further study. There are also a lot of factors that may contribute to productivity that were not covered in this study to keep the scope manageable. Some study into their effects may also be warranted.
Examples of subjects that may be further looked into are:

- This research failed to show a concrete correspondence between centralization and productivity. Further research might look into the possibility that centralization exerts some other influence on either productivity.
- Other aspects of the friendship graphs other than the three above might be looked into, for example the connectivity of the friendship graphs.
- Other attributes than the six investigated in the second part of the study may be investigated.

REFERENCES

Berman, E. M., West, J. P. and Richter Jr, M. N. 2002. Workplace relations: Friendship patterns and consequences (according to managers). ***Public Administration Review***, 62(2):217–230.

Cohen, S. G. and Bailey, D. E. 1997. What makes teams work: Group effectiveness research from the shop floor to the executive suite. ***Journal of Management***, 23(3):239–290.

Crampton, S. and Mishra, J. 1999. Women in management. ***Public Personnel Management***, 28(1):87–107.

Danowski, J. A. 1980. Group attitude uniformity and connectivity of organizational communication networks for production, innovation, and maintenance content. ***Human Communication Research***, 12:251–270.

Dearborn, R. and Simon, H. 1958. Selective perceptions in executives. ***Sociometry***, 21:140–144.

Jehn, K. A. and Shah, P. P. 1997. Informal relations and task performance: An examination of mediating processes in friendship and acquaintance groups. ***Journal of Personality and Social Psychology***, 72(4):775–790.

Kratzer, J., Leenders, R. T. A. J. and van Engelen, J. M. L. 2004. Informal contacts and performance in innovation teams: The important role of 'family ties'. ***International Journal of Manpower***, Accepted for publishing.

Matheson, C. 1999. The source of upward mobility within public sector organizations. ***Administration and Society***, 31(4):495–525.

Rawlins, W. K. 1983. Negotiating close friendship: The dialectic of conjunctive freedoms. ***Human Communication Research***, 9:255–266.

Roloff, M. E. 1987. Communication and reciprocity within intimate relationships. ***Interpersonal Processes: New directions in Communication Research***.

Shirase, K., Nagafune, N., Wakamatsu, H. and Arai, E. 2000. Human oriented production management considering working satisfaction. ***Proceedings of Pacific Conference on Manufacturing***, 9:733–738.

Simonetti, J. and Ariss, S. 1999. Through the top with mentoring. ***Business Horizons***, 42(6): 56–63.

Wilensky, H. 1967. ***Organizational intelligence***. New York: Basic Books.

Zaccaro, S. J. and Lowe, C. A. 1986. Cohesiveness and performance on an additive task: Evidence for multidimensionality. ***Journal of Social Psychology***, 128:547–558.

The Effects of Age, Experience and Tenure on Team Creative Performance

Tjaard de Vries and Mark Bastiaans (University of Groningen)
t.de.vries.9@student.rug.nl, m.bastiaans@student.rug.nl

This study focuses on the effects of different age, experience and tenure factors on creative team performance. After quantitive single-level and multi-level regression analysis of data collected on a multitude of teams in different organisations with a strong research and development focus, we arrive at some interesting conclusions, one of which is that a higher average team age leads to a higher creative level. We argue that a well-mixed research and development team in terms of these factors generally leads to improved creativity.

Everyone respects the elders in a community. Everyone respects and admires their wisdom and insights. Elders let us see things in a way we wouldn't have thought of ourselves. Why? They have vast amounts of experience in their field through years and years of practice. Older, more experienced people form the foundation of any team, relieving their team members of the burden that one encounters when exploring new territories in one's career.

But is this the only factor in deciding a team's success? A Research and Development Team in a corporate environment does not exclusively rely on experience to achieve the greatest of results. Performance relies heavily on creative capabilities. To beat all the competitors and to even stay one step ahead of them all, a team must be creative as well, in order to deliver new and innovative products, greatly exceeding the market's imagination.

Experience and age on the one side, creativity on the other. Do these two sides mix? Today's managers and corporate executives are extremely interested in how the average age, the amount of service years with the company, and the number of experience years affect a team's creativity. In this world of decreasing time-to-market and ever increasing competition and consumer demands, one must maximize team performance in every possible way. Ensuring that a team's creativity level is high is one way to assure that ideas that appeal to the masses are implemented efficiently.

The goal of our research is to help managers effectively base their Research and Development teams' composition on experience and age. In order to do this, we have acquired data on a large amount of innovation team members and project teams in corporate environments.

Theory and Hypotheses

Our analysis focused on four main factors on the team level: a team's creativity level, measured by several sub-factors such as how team members experience creativity, and the average age, the average team member's experience in his or her field of expertise, and the average amount of years the team members have worked for their current employer. We will use quantitative based analysis methods to try to make accurate statements about our research question.

First of all, we will provide the variables that are relevant to our research question and that may mutually influence each other. We need to do so because otherwise no meaningful conclusions can be made from the research. The variables are:

- Age of team members
- Age diversity within a team
- Age of the team (i.e. how long the members have been together)
- Experience of team members in the field
- Tenure

Our main research question is:

"How do a team's age, experience and service years affect a team's creativity level?"

Together with the list of variables, this question leads to the following hypotheses:

H1: *The younger the average team member, the more creative the team is.*

Though it may sound trivial, this hypothesis surely needs checking because it may be just scratching the surface of the actual problem. Our society is quite general in dividing the masses into age groups. Society's view of age is that younger people tend to be more enthusiastic and display a more adventurous way of getting to their goals. Young adults are prominent in war, revolution, immigration, urbanisation and technological change (*The cohort as a concept of social change, N.B. Ryder,* 1965). Older people tend to have a more conservative view of things and are trapped within their own routines. However, since older people tend to think that they are behind on their career track, they could put more effort into their work and thus increase their creative output (*Age Grading: The Implicit Organizational Timetable, B.S. Lawrence, 1984*).

H2: *A high age variance within a team increases the team's creativity level.*

Reading *The cohort as a concept in the study of social change (N.B. Ryder, 1965)*, one might expect that this hypothesis partially depends on H1. In fact, it probably does, as (according to this article) older persons tend to be at the top of a social hierarchy, which may enhance their presumed creativity inhibiting tendencies. However, one shouldn't forget that people learn from each other and that the younger members may be an inspiration to their older peers which in essence could lead to an efficient environment, in which the experienced members provide a playground in which the younger ones can fully expose their ideas.

H3: *The creativity of a team decreases with the time the team is together.*

Again, this hypothesis resembles H1 quite a lot, but this time the team as a whole is viewed. There are quite a lot of parallels that can be seen between the behaviour of an individual and the behaviour of groups. Despite the diversity of a group, the group as a whole can be said to have an opinion on things, to have a favourite pastime, etc. One of these things is the age of a group and the ways of acting that have evolved during the team's existence. This can vary from communication to working routines, and we suspect the latter to have a negative effect on the team's creativity, for reasons similar to those mentioned in H1. (Also see: *The black box of organizational demography, B.S. Lawrence, 1997*).

H4: *Experience of team members improves team creativity.*
H5: *Experience of team members inhibits team creativity.*

Experience is a factor that bears a certain duality; hence these two hypotheses will be explained together. On one hand, one can view experience as an ever expanding thing that grows during one's life. Experience alters one's way of thinking and may stimulate one's imagination. On the other hand however, experience has quite a few resemblances to age, especially the way in which daily routine can be devastating one's will to explore; at least an experienced person is expected to display a certain arrogance which narrows his view and thus has a negative effect on his creativity.

H6: *Greater tenure inhibits team creativity.*

Although this may seem a copy of H3, it isn't. Of course it depends on the company's size, but companies usually have standardized work ethics and ways of dealing with various things. One can shift teams as much as one likes, chances are that the employees will still have the same boss, still look at the same kind of computer screens in the same kind of rooms and eat with the same people in the same canteen as the did before.

Though a new team may inspire an employee, the boredom that comes with the years and years of working in the same company surely will affect creativity, as these employees may be greatly inspired with renewed energy and creativity as soon as they apply to a new job at the competitor. Though it is beyond the scope of this paper, corporate executives may consider diversifying the company structure among different locations to encourage employers not to leave if this hypothesis is accepted.

The Data

The data set, acquired from *Communication and Performance: An Empirical Study in Innovation Teams (J. Kratzer, 2001)*, contains data acquired over 5 periods of data gathering over 44 Research and Development teams from a total of 11 companies. The data consists of answers from individual team members on a questionnaire adapted from the well-known Stanford questionnaire described in *The Stanford Health Assessment Questionnaire: Dimensions and Practical Applications* (B. Bruce, J. F. Fries, 1978). We only require specific parts of the data, namely:

- **TNR**: Team Number Identification. We use this number to distinguish between different teams, since the research is on the team level, not on the individual level,
- **V1**: Number of team members. We use this to calculate averages for other data fields.
- **V10**: Perceived Team Creativity. Team members were asked about their view on their team's creativity level. This level has a scale from 1 to 7, 1 being worse than average and 7 much better than average. In general, teams have a pretty good idea about how they themselves are functioning, so this piece of data is a good indicator of team creativity.
- **V15**: Years working in field of specialization.
- **V16**: Tenure in current company (in years).
- **V17**: Tenure in current team (in years).
- **V21**: Age. The age is measured in five groups, group 1 being under 30, 2 being 30-39, 3 being 40-49, 4 being 50 to 59 and 5 being 60 and up.

Methods

We used the data described to prove or disprove our hypotheses. Each hypothesis requires different data, so we will describe the data used for every hypothesis we postulated. In addition to this, we will describe the way we will use the data to prove or disprove the hypotheses using single-level regression analysis.

H1

We defined H1 as: "The younger the average team member, the more creative the team is". This obviously points to age (V21) and perceived creativity (V10) as the main variables. In addition, we need to calculate the mean age and perceived creativity, which we will do for each team using their TNR and V1. We will then compare the mean age per team and the mean creativity level per team using a scatter plot.

H2

We defined H2 as: "A high age variance within a team increases the team's creativity level". For this, we calculate the age variance in each team and again the mean perceived creativity level per team. We then compare these using the same methods as those for H1.

H3

H3 is defined as: "The creativity of a team decreases with the time the team is together". In this case, we first need to calculate the average team tenure for each team (V17) and plot this together with the creativity level in a manner similar to the method used in H1 and H2.

H4, H5

H4 and H5 are defined as "Experience of team members improves/inhibits team creativity". These hypotheses will be tested at once and depending on the results one of these will be accepted (or both will be rejected if no correlation is found at all). The variables used are the mean experience for each team (V15) together with the perceived creativity. Again, things will be analyzed using a scatter plot.

H6

This hypothesis is defined as "More service years at the company inhibits team creativity". This is our last hypothesis to test and we will be using the company tenure (V16) and the perceived creativity. As in the previous hypotheses, these data will be plotted in a scatter plot.

Now that we have described in which way we have analyzed the actual data, we analyze the actual data itself. We will do this using simple regression analysis. In addition, we will use a multi-level analysis to prove relationships between the different factors.

Results

For doing simple regression analysis on the data, we used Microsoft Excel, the spreadsheet part of Microsoft Office, which can perform basic statistical tasks and generate basic tables and charts.

H1

The scatter plot for H1 shows a lightly growing linear correlation between the average age group and average perceived creativity. This actually signifies that H1 is not true.

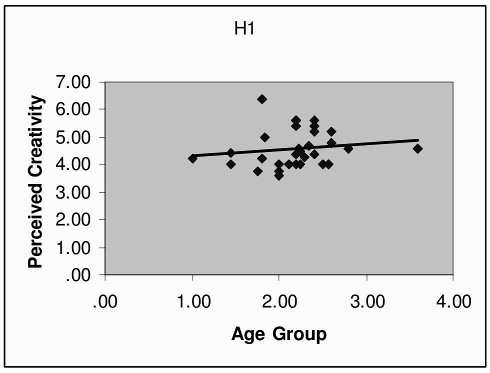


Figure 1: Scatter Plot for H1

H2

There seems to be a constant relationship between age group variance and average perceived creativity. This means that H2 is not true, although age variance seems to be low in general.

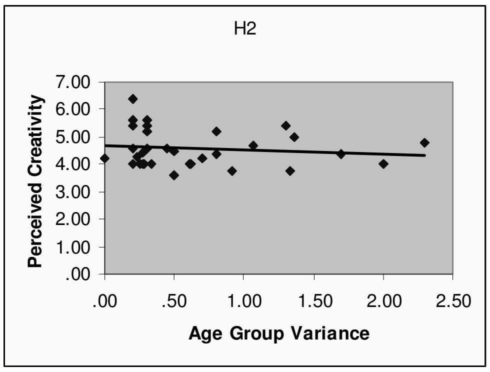


Figure 2: Scatter Plot for H2

This means that our chosen test leaves some accuracy to be desired. A larger sample will undoubtedly be more accurate and provide for smaller error margins.

H3

As expected, a clear negative linear correlation exists between team tenure and perceived creativity. This proves H3.

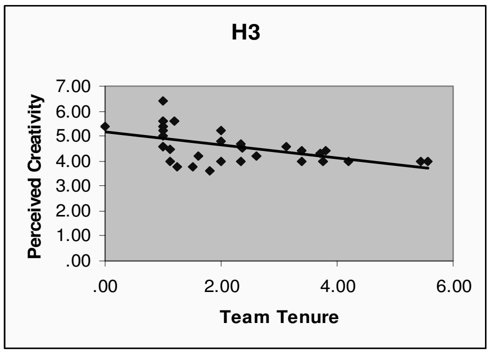


Figure 3: Scatter Plot for H3

H4, H5

There seems to be a constant relation between team experience and perceived creativity. Therefore, neither H4 nor H5 hold.

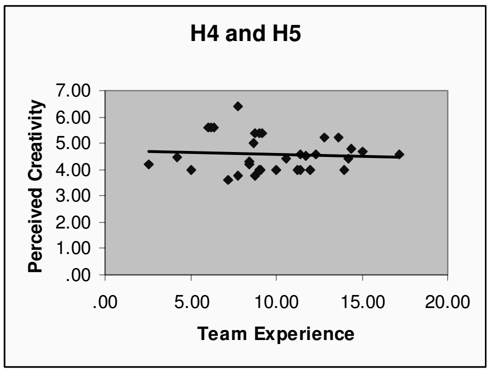


Figure 4: Scatter Plot for H4, H5

H6

The scatter plot for H6 shows a slightly negative correlation between average company tenure and perceived creativity. Therefore, H6 holds.

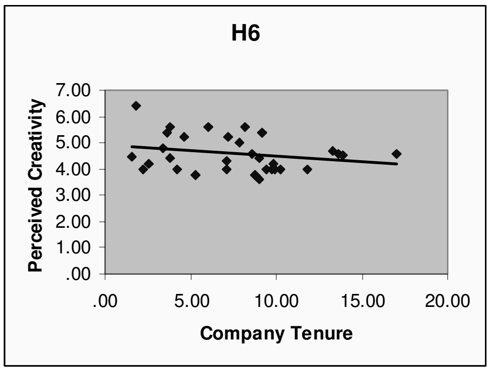


Figure 5: Scatter Plot for H6

Multi-Level Analysis

A single-level analysis of the data is usually not sufficient to show the relative influence of the different factors. Put differently, we wish to show if, for instance, age, has a greater effect on team creativity than team tenure. Therefore, we decided to try the hypotheses by doing a multi-level analysis (*Multivariate Analysis Techniques in Social Science Research*, Tacq, J. J. A., 1997).

Such an analysis will provide for a regression formula which, in addition to providing information on the influence of the individual factors on average team creativity, accounts for correlation between the different factors.

This analysis was done by using MLWin, a program tailored for doing such tasks. After entering the data and calculating the team averages, the following formula was estimated:

$$\begin{aligned}
\boldsymbol{y}_{ijkl} &= 4.150(0.197) \\
&- 0.008(0.021)\ \boldsymbol{i}_{ijkl} \\
&- 0.049(0.012)\ \boldsymbol{j}_{ijkl} \\
&- 0.250(0.031)\ \boldsymbol{k}_{ijkl} \\
&+ 0.654(0.136)\ \boldsymbol{l}_{ijkl} \\
&+ \boldsymbol{e}_{ijkl} \\
\boldsymbol{e}_{ijkl} &\sim \mathbf{N}(0, \boldsymbol{\sigma}_e^2) \\
\boldsymbol{\sigma}_e^2 &\sim -0.255(0.026)
\end{aligned}$$

The independent variables **i**, **j**, **k** and **l** represent the averages of the years in field, company tenure, team tenure and age, respectively. The dependent variable **y** denotes the average team creativity and **e** the error rate. The numbers noted before the independent variables are their standardized beta coefficients and standard deviations.

The above numbers show the relative importance of the different independent variables. Age seems to be the biggest contributor to the dependent variable. The runner-up is team tenure, while the first two factors contribute only minimally.

For the results to actually have some meaning, we take an error margin of 10 percent into account. We take such liberties because of the size of the data set: a mere 192 cases were taken into account for the formula. The **σ** is an indicator for the error rate: by dividing the beta coefficient (0.255) with the standard deviation (0.026), an error rate of less than 10 percent is derived.

The results again disprove H1, because age has a relatively high influence on team creativity. H2 is not examined. H3 is proven also, while H4 and H5 again do not hold due to the very slight influence of team experience. H6 holds again because of the relative influence of company tenure. However, this influence seems to be very slight.

Now that we have performed both analyses and examined the results, we can arrive to conclusions.

Conclusions and Recommendations

After simple data analysis, H3 and H6 were proven true, while he other hypotheses were not so lucky and proved false. We therefore arrive to the following conclusions:

1. The creativity of a team does not decrease as the average age increases. More to the opposite: it actually increases! It seems that older team members have a positive influence on team creativity.
2. In terms of creativity, it does not matter if the age of team members differs greatly. Therefore, this isn't really an issue to worry about when composing project teams.
3. Teams that are together for an extended period of time seem to be less creative than teams that just got together. This could be an incentive for management to change team composition once in a while.
4. The average team experience does not affect creative performance at all.
5. Opposed to age, company tenure does decrease team creativity.

From these conclusions, some recommendations to managers of research teams can be made. First of all, a well-mixed team in terms of age, tenure and experience is generally a good idea to begin with. But for elaboration, the results of the multi-level analysis should be used.

Multi-level analysis showed the relative influence of all the factors on creative team performance. By using these relative factors, we conclude that:

1. Average team age is the most important factor in the total equation for team creative performance,
2. Team tenure is, in effect, the second most important factor, and
3. Average company tenure and years in the field of specialization do not influence team creativity greatly.

Therefore, the ideal innovation team is a team with a high age average for boosting the team's creative output. Furthermore, one should change team composition often to keep people innovating, and to encourage creativity in one's company.

Any further recommendations that can be made depend on what is expected of the teams. Should one want one outstanding team for a particular prioritized project, one should make a fresh team of somewhat older team members for the task, both because of the increase of creativity with age and because of the slight decrease with age variance. The teams composed of the remaining employees might suffer a bit though, and if the company doesn't hire and/or fire many employees the effect of this trick might wear off after a while because increasing team tenure. However, it might be a good idea to prioritize older applicants if any vacancies within the company occur.

These conclusions and their derived recommendations are beneficial for managers and team members alike. Managers can compose their teams for better creative results by using the results of this study, while older employees will find it comforting that one argument used for laying off older employees, namely that of that older people have less creative influence on a team than younger people, appears to be a persistent urban legend that needs urgent revision.

We note that the scope of this study was on a team level. Since teams are composed of quite a few individuals, some more elaboration on the effect of those individuals would be in order. We leave this analysis to a following study.

References

1. *The cohort as a concept of social change*, N.B. Ryder, 1965
2. *Age Grading: The Implicit Organizational Timetable*, B.S. Lawrence, 1984
3. *The black box of organizational demography*, B.S. Lawrence, 1997
4. *Communication and Performance: An Empirical Study in Innovation Teams*, J. Kratzer, 2001
5. *The Stanford Health Assessment Questionnaire: Dimensions and Practical Applications*, B. Bruce, J. F. Fries, 1978
6. *Multivariate Analysis Techniques in Social Science Research*, Tacq, J. J. A., 1997

TEAM CREATIVITY: INFLUENCE OF HETEROGENEITY IN AGE, SEX, EDUCATION AND SERVICE YEARS ON CREATIVITY OF R&D TEAMS

FRANK VAN DEN NIEUWBOER
frankie@fmf.nl
KLAAS-JAN STOL
kjs@fmf.nl

Department of Computing Science
University of Groningen

The purpose of this study was to examine the relationship between team diversity and experienced team creativity. In order to measure team diversity, we looked at four types of diversity. These were diversity of sex, diversity of education level, diversity of age and diversity of the number of service years. Data was gathered from 33 R&D teams.
The results showed that diversity of sex has a slight positive effect on the team creativity. The other attributes showed there could have been a relation, but this was not significant.

In order to survive the always present competition, a company should distinguish itself, so it can offer unique services or products. If a company is not able to distinguish itself among the numerous other companies offering the same services or products, it is not unlikely the company will be put out of business.

Therefore, innovation and creative minds are very important to a company. This is especially true for companies with R&D teams. The purpose of an R&D team is, after all, to innovate.

How an R&D team is functioning depends on a number of factors. A very important one is, of course, what persons the team consists of. If the members of a team all have creative mind sets, then this team is likely to function better than a team consisting of members who are not as creative. However, it is not as easy as calling one team ”creative” and yet another as not being creative. This creativity of a team is of course, dependent on the individual members, but also on the team as a whole. Creativity is just one aspect of a member's personality. It may well be that she[1] feels not comfortable to express her ideas. This way, the team as a whole does not take advantage of the creativity of all members.

In order to prevent this, all team members should be comfortable being a member of the team. However, it is not clear in advance when this will be the case. So, the question for a manager is how to compose her team, so that it will exploit the

[1] *she* should be read as *he or she*.

creative minds of all individual members. When creativity is higher, solutions to problems will be thought of faster and more easily. This in effect reduces the costs of development of products, and therefore is very important to a company as a whole.

Of course, it is interesting to know what makes a team more creative. In recent years, some research has been done on the influences on team performance. Timmerman (2000) researched the influence of racial diversity and age diversity on team performance. Cady and Valentine (1999) investigated the influence of team diversity on quality and quantity of innovation. This diversity was measured in diversity of age, sex, race and function.

Other researchers concentrate on a specific property, such as sex of team members (Schruijer & Mostert, 1997). However, that was measured on an individual level. In this article, we take a broader view on diversity, and we will compare diversity at team level. We will consider heterogeneity of age, sex, the number of service years and the level of education of the members of an R&D team.

RESEARCH

Some project teams show a higher level of creativity than others. The question is, how one can influence the creativity of project teams, so managers can compose teams that will be more creative. Therefore, we propose the following research question:

What is the effect of team heterogeneity on creativity of R&D teams?

Because heterogeneity is very broad, we will focus on only four attributes of R&D teams. We will investigate what the specific effects are of heterogeneity of *age*, *sex*, *number of service years* and the *level of education* of members of R&D teams on creativity. Therefore, we propose the following subquestions:

- *What is the effect of diversity of sex on the creativity of R&D teams?*
- *What is the effect of heterogeneity of age on the creativity of R&D teams?*
- *What is the effect number of service years on the creativity of R&D teams?*
- *What is the effect of the level of education on the creativity of R&D teams?*

With these subquestions, we inspect four very important attributes of teams. This way, we can get a good idea of what effect heterogeneity in team composition has on the creativity of R&D teams.

The second research question will then concern managers, so we propose the next research question:

How should a team manager compose her R&D team?

THEORY AND HYPOTHESES

Heterogeneity of Sex in R&D Teams

Schruijer and Mostert (1997) showed that diversity of sex in brainstorming groups had a positive influence on creativity. According to their study, members engaging in brainstorming in heterogeneous groups generate more ideas, associations and angles than those in homogeneous groups. Also, individuals in heterogeneous groups rated the process as a more positive one. This indicates that members in a heterogeneous group are more comfortable to express their ideas and propositions. As we suggested in the introduction of this article, this is important for team creativity as a whole.

Therefore, we propose the following hypothesis:

Hypothesis 1: Diversity of sex has a positive influence on creativity of R&D teams.

Heterogeneity of Age in R&D Teams

Heterogeneity of age is also an important factor when considering creativity of teams. Although not much research relating to age and creativity has been conducted, a study relating to age and performance (Pelled et. al., 1999) showed that homogeneous groups had more emotional conflict than heterogeneous groups. This was explained by suggesting that age is used for comparing one's accomplishments and career progression. So, one could suggest that when members of a team have more or less the same age, they try harder to distinguish themselves.

Other research relating to the effects of age on innovation suggests that there are few significant effects of age heterogeneity (Wiersema & Bantel, 1992; Zenger & Lawrence, 1989).

Therefore, we present the following hypothesis:

Hypothesis 2: Heterogeneity of age has a negative influence on creativity of R&D teams.

Heterogeneity of service years in R&D Teams

Not much information can be found on research relating to the number of service years a person has, and his or her level of creativity. However, while people with a longer career at a particular company may suffer of boredom, they do have experience, and know how to solve a great number of problems. They already learned how to think of solutions.

On the other hand, while people who have less service years at a company do not have this experience, they will suffer less of boredom because many tasks and problems may come as new to them. Also, they may be more motivated, because they do not have the advantage of "being there too long". We think a combination of people with a lot of experience, and people that are still *fresh-minded* may help in the level of creativity that an R&D team exposes. Therefore, we propose the following hypothesis:

Hypothesis 3: Heterogeneity of service years has a positive influence on creativity of R&D teams.

Heterogeneity of education level in R&D Teams

As with the previous section, not much information on research relating to the level of education and creativity can be found. However, one could argue that people on a 'different level' have different mind sets, and thus are likely to have other approaches to tackle problems. Therefore, we feel that a team with more variety in levels of education is likely to have more different approaches to find solutions, and thus the group as a whole has more creative potential. Therefore, we propose our last hypothesis as follows:

Hypothesis 4: Heterogeneity of education level has a positive influence on creativity of R&D teams.

METHODS

Participants

All participants in the research are members of Research and Development (R&D) teams. The participants were asked to "asses the state of the art of their innovation teams" (Kratzer, 2001). Data for our research was gathered from 33 R&D teams, involving 199 team members in total, at 11 companies.
The data was built upon a Stanford University questionnaire, which had been adapted to better suit the research needs. Data was gathered in five phases. These are summarized below.

1. In phase one 39 companies were selected based on their innovation methods.
2. In the second phase the directors of these companies were sent an invitation letter containing a summary of the main questions.
3. In the third phase presentations were given at 21 companies.
4. In the fourth phase the action plan for the questionnaires was composed, and the questionnaire was done.
5. In the last phase the collected data was processed into a "team profile" and suggestions were delivered to these 39 companies.

Measures

Measuring Team creativity Participants were asked to consider the team as a whole. For measuring the team creativity, the team score was on a seven point scale of 1 (very bad) to 7 (very good). Then, we take the average score of all members in a group.

Measuring Sex Heterogeneity For measuring sex heterogeneity, we determine the number of men and the number of women in each team. Next, we count the number of heterogeneous pairs. That is, a pair consisting of one man and one woman. Then, we divide this number by the total number of members divided by two. This is, of course, the total number of pairs, neglecting the fact that not all "pairs" are heterogeneous. The outcome is a number between 0 and 1, so multiplying this by 100% results in a percentage indicating the heterogeneity of sex.

Measuring Age Heterogeneity Participants were asked in which age class the belonged to. The following age classes were defined:

- under 30
- 30-39
- 40-49
- 50-59
- over 60

It would not be correct to just measure the average score, because if the team consisted of two equally sized groups, for example one group in class **under 30**, the other in class **over 60**, then that would result in the same average as when all members were equally divided over all five classes.

Therefore, it was necessary to come up with an alternative measure. We describe this alternative measure below.

To determine the age diversity score we first have to calculate the highest deviation of the perfect diversity score (the case when all classes are equally sized). This can expressed by the following formula:

$$M = \frac{N}{5} \times 4 \times 2$$

With M being the maximum deviation, when all members belong to one class; N being the number of team members.

The constant 5 stands for the number of choices, and the multiplication by 4 is used because in the case of total homogeneous partition in four classes the absolute deviation will be $\frac{N}{5}$ in each class. The multiplication by 2 is performed because the deviation is accounted for twice. The value of M denotes the maximum number of diversity.

In the case of perfect partition, one would expect a mean value of $\frac{N}{5}$ members in each class. To determine the actual deviation, we calculate for each class the absolute deviation with respect to the mean value. So, to calculate the actual deviation, we use the following formula:

$$D = \sum_{i=1}^{5} \left| class_i - \frac{N}{5} \right| \tag{1}$$

Where $class_i$ is the number of members in class i.

Finally, to calculate the percentage of age diversity we have to calculate the last step, as shown in formula (2).

$$A = \frac{M - D}{M} \times 100\% \tag{2}$$

With A being the percentage of age diversity in the R&D team.

A percentage of 0 effectively means that $M = D$. In other words, the actual deviation equals the maximum deviation, which only occurs when all members belong to one class. So, in that case, there is no diversity, which mirrors the zero percentage.

On the other hand, a percentage of 100 means that D is 0. This means that the actual deviation, calculated by formula (1) is 0. This is only the case, when all classes contain the *expected* or *mean* number of members. This is only the case if there is a perfect partition, in other words, a complete heterogeneous group (when speaking of age).

Measuring Heterogeneity of service years

Diversity of service years was measured as follows. All participants were asked what their number of service years with the company was. To express the diversity of number of service years within a team, we take the **standard deviation** as a measure, denoted by σ. If σ is high, then there is more diversity than when σ is low.

Measuring Heterogeneity of education level

Diversity of education level was measured as follows. All participants were asked what their *highest* degree they received was. The answers were partitioned into six classes. These six classes were:

1. elementary education
2. secondary education
3. higher secondary education
4. polytechnic education
5. academic education/university
6. Ph.D.

To calculate the diversity percentage, we used the same formulae as for calculating the diversity percentage for age. Of course, instead of five classes, we now have six.

Analysis

We used both single-level modeling and two-level modeling. For each hypothesis we made a separate model, and we tried to find an equation for the regression line. We did this at both single and multi level.
In our single-level model, observations of the teams members are middled and taken into account for the determination of the equation.

In our two-level model, observations of the individual team members are the lower level observations. The observations of the teams are the higher level observations. This two-level model is necessary because observations within a team are not independent.

For each hypothesis we compared results by using the $-2 \times LogLikelihood$ both at single and at multi level. The difference between these two results is compared to a χ^2-distribution with 1 degree of freedom. This was to validate the results found in the two-level observation for significance.

Missing data

Of course, whenever research is done with use of data gathered from people, there is the possibility of missing data. That is, on some questions not all participants gave an answer. For that problem we use a simple solution. When a team member did not answer the question, then that member was not taken into account. Effectively, the team consisted only of members who answered all questions. In our opinion, this solution to the problem of missing data is reasonable, because we do not consider team *size*.

RESULTS

All results can be found in appendix A. Based on the results in table (9), one cannot easily see if there is any correlation whatsoever between the four measures and the level of creativity. For that purpose, we will now include four different plots, to check for any correlation.

Heterogeneity of Sex and Creativity

In figure (1) we can see that there is no correlation between the level of diversity in sex and the level of creativity that is experienced. On a qualitative level we cannot

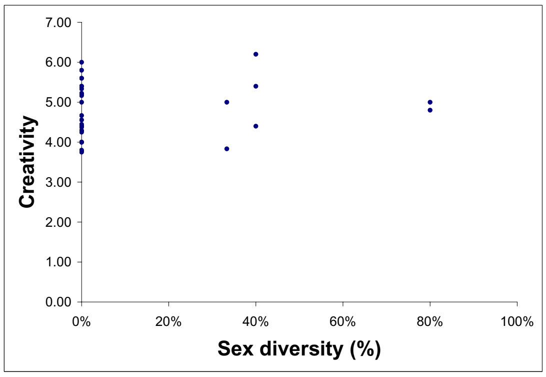


Figure 1: The correlation between diversity in sex and creativity.

find any relationship between gender and creativity. This is also due to the fact that there are only seven teams with one ore more female members. Figure 1 shows a scatter plot of the results. To further investigate this, we use multi-level analysis. The results of this can be found in table .

FIXED EFFECTS	
baseline team creativity	3.959 (0.334)
Gender	0.566 (0.309)
RANDOM EFFECTS	
team-level variance σ^2	0.990 (0.101)
-2 $\times$Log Likelihood model	542.91

Table 1: Results of single level regression on heterogeneity of gender in R&D teams

The overall mean level of creativity considering sex is a score of 4.557. As we can see in the resulting model, the mean difference in creativity scores can be ascribed to sex by 0.460 times the standard deviation. The variance partition coefficient is calculated as follows:

$$\frac{0.304}{0.304 + 0.705} = 0.301$$

This means that about 30 percent of the variance in creativity considering sex can be ascribed to variations between teams.
The difference between both $-2*$Log Likelihood models can be calculated simply:

$$542.91 - 518.74 = 24.17$$

FIXED EFFECTS	
baseline team creativity	4.557 (0.116)
Gender	0.460 (0.303)
RANDOM EFFECTS	
team-level variance σ^2	0.304 (0.107)
member-level variance σ^2	0.705 (0.079)
-2 $\times$ Log Likelihood model	518.74

Table 2: Results of multi-level regression on heterogeneity of gender in R&D teams

Using the χ^2-test on this results (1 degree of freedom) shows us that this solution is significant.

This shows us that there is a relation between heterogeneity of sex and the experienced team creativity. At some level, the overall creativity is positively dependent on heterogeneity. When we use a t-Test with ($p = 0.01$) we can see this relation is significant.

Diversity of Age and Creativity

In figure (2) we can see that there is no correlation between the level of diversity of age and the level of creativity that is experienced. However, it may be interesting to

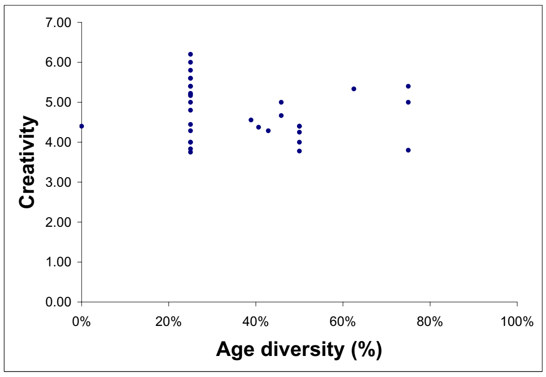


Figure 2: The correlation between diversity age and creativity.

further analyze this data. We do this by means of a box plot diagram. In figure (3) such a box plot diagram is shown. The data was divided into two classes: one class containing teams with a low age diversity score, another class containing teams with a higher age diversity score. If the diversity score of a team is less than 35%, the team belongs tot the lower class, otherwise the team is counted in the other group.

In figure (3), it can be seen that the class with lower age diversity has a higher creativity score. On the other hand, the class with a higher age diversity has a

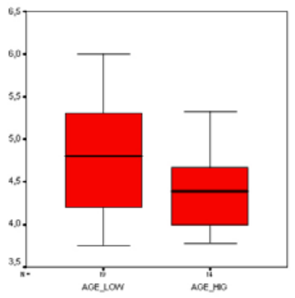


Figure 3: Partition into a group with low age heterogeneity and a group with high age heterogeneity.

significant lower creativity score. This confirms our hypothesis, which claimed that diversity of age in an R&D team has a negative impact on the level of creativity.

It should be kept in mind, that this creativity score was obtained by asking people about their own experience. That is, a high creativity score does not necessarily mean that the actual creativity is high.

FIXED EFFECTS	
baseline team creativity	4.540 (0.202)
Age	0.008 (0.087)
RANDOM EFFECTS	
team-level variance σ^2	1.007 (0.103)
-2 $\times$ Log Likelihood model	546.229

Table 3: Results of single level regression on heterogeneity of age in R&D teams

When we use the the same data with multi-level regression both at team and individual level, we get the following results:

FIXED EFFECTS	
baseline team creativity	4.695 (0.214)
Age	-0.049 (0.082)
RANDOM EFFECTS	
team-level variance σ^2	0.321 (0.111)
member-level variance σ^2	0.708 (0.079)
-2 $\times$ Log Likelihood model	520.67

Table 4: Results of multilevel regression on heterogeneity of age in R&D teams

The overall mean level of creativity considering age is a score of 4.695. As we can see in the resulting model, the mean difference in creativity scores can be ascribed to age by -0.049 times the standard error, which is rather low.

$$\frac{0.321}{0.321 + 0.708} = 0.312$$

This means that about 30 percent of the variance in creativity considering age can be ascribed to variations between teams.
The difference between both $-2\times$Log Likelihood models can be calculated simply:

$$546.229 - 520.67 = 25.559$$

Using the χ^2-test on this results (1 degree of freedom) shows us that this solution is significant.

Although the two-level model is significant, we cannot see that creativity is dependent of heterogeneity of age. At single level we see there is a very slight positive impact (+0.008), but at two-level regression we see a negative impact (-0.049). Using a $t-$Test we can see this relation is not significant ($p = 0.01$).

Diversity of Service years and Creativity

In figure (4) we can see that there is no clear correlation between the number of service years and the level of creativity that is experienced. However, when we draw the trend line we can see a slight negative relation, this indicates that diversity of service years has a slight negative impact on the experienced level of creativity.

This might be explained by the behavior of all members. The younger people may have respect for and expectations from the people with longer service time. So, they might think that *the other members* will know better. So, they will not mention any new or unusual ideas of themselves, because they are afraid not to be taken seriously.

On the other hand, the members with longer service time may have expectations of their own. They may have been looking forward to a 'new wave' of ideas from the new members. So, they give the new members space to propose their ideas.

This situation could be created by the fact that both 'new' members as well as members with longer service time have opposite expectations from 'the other' members.

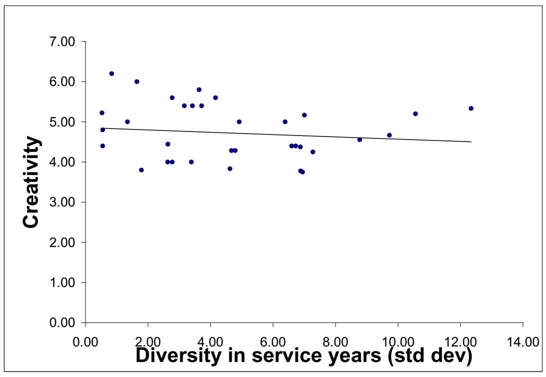


Figure 4: The correlation between diversity in the number of service years and creativity.

Now when we apply multi-level regression we get the following results:

FIXED EFFECTS	
baseline team creativity	4.713 (0.112)
Service years	-0.020 (0.011)
RANDOM EFFECTS	
team-level variance σ^2	0.990 (0.101)
-2 $\times$ Log Likelihood model	542.957

Table 5: Results of single level regression on heterogeneity of service years in R&D teams

FIXED EFFECTS	
baseline team creativity	4.715 (0.141)
Service years	-0.017 (0.011)
RANDOM EFFECTS	
team-level variance σ^2	0.302 (0.106)
member-level variance σ^2	0.706 (0.079)
-2 $\times$ Log Likelihood model	518.68

Table 6: Results of multi-level regression on heterogeneity of service years in R&D teams

As we can see in the resulting model, the mean difference in creativity scores can be ascribed to service years by -0.017.

The variance partition coefficient is calculated as follows:

$$\frac{0.302}{0.302 + 0.706} = 0.300$$

This means that about 30 percent of the variance in creativity considering service years can be ascribed to variations between teams.
The difference between both -2×Log Likelihood models can be calculated simply:

$$542.957 - 518.68 = 24.295$$

Using the χ^2-test on this results (1 degree of freedom) shows us that this solution is significant. We can see at both at single-level (-0.020) and at two-level (-0.017) heterogeneity of Service Years has a slight negative impact on the experienced team creativity. Using a $t-$Test we can see this relation is not significant ($p = 0.01$).

Diversity of Education and Creativity

In figure (5) we can see that there is no correlation between the level of diversity in education and the level of creativity that is experienced. However, figure (5) we expect there might be some connection between diversity of education and creativity, because the number of teams is clustered somehow in the middle of the figure. To better determine the relation between diversity of education and creativity we could draw a box plot. This is done in figure (6).

We can make a division into three clusters. The first cluster would contain all teams with an education diversity score below 25%, the second containing teams with a score between 25% and 35% and the third cluster containing teams with a score higher than 35%. In figure (6) this division into three groups is shown by

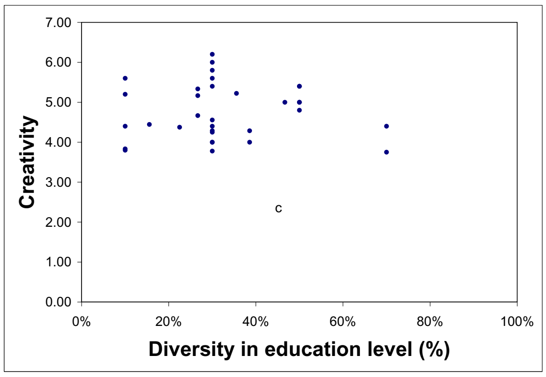


Figure 5: The correlation between diversity in education and creativity.

means of a box plot diagram. In the box plot figure (figure (6)), we can see that

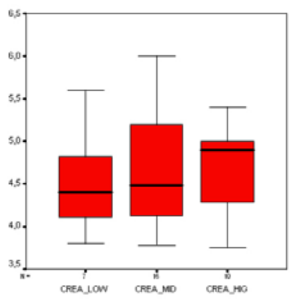


Figure 6: The correlation between diversity in education and creativity.

the two clusters with a higher level of diversity in education level do in fact have somewhat higher scores. However, the lower bound is still the same as the lower cluster.

Considering this, one could say education diversity has to a certain extend a positive influence on the experienced creativity of a Research and Development group. But when educational diversity becomes too high, this impact decreases somewhat.

FIXED EFFECTS	
baseline team creativity	3.705 (0.380)
Education	0.195 (0.085)
RANDOM EFFECTS	
team-level variance σ^2	0.980 (0.100)
-2 $\times$ Log Likelihood model	541.084

Table 7: Results of single level regression on heterogeneity of education in R&D teams

When we apply multilevel regression both at team and at individual level we get the following results:
The overall mean level of creativity is a score of 4.407, with a standard error of

FIXED EFFECTS	
baseline team creativity	4.407 (0.415)
Education	0.041 (0.091)
RANDOM EFFECTS	
team-level variance σ^2	0.300 (0.106)
member-level variance σ^2	0.715 (0.080)
-2 $\times$ Log Likelihood model	520.85

Table 8: Results of multilevel regression on heterogeneity of education in R&D teams

0.415. As we can see in the resulting model, the mean difference in creativity scores can be ascribed to education by 0.041 The variance partition coefficient is calculated as follows:

$$\frac{0.300}{0.300 + 0.715} = 0.296$$

This means that about 30 percent of the variance in creativity considering education can be ascribed to variations between teams.
The difference between both $-2\times$Log Likelihood models can be calculated simply:

$$541.084 - 520.85 = 20.234$$

Using the χ^2-test on this results (1 degree of freedom) shows us that this solution is significant. At single level heterogeneity of Education has a positive impact on experienced team creativity (+0.195), but at second level regression this is (+0.041). Using a $t-$Test we can see this relation is not significant ($p = 0.01$).

CONCLUSIONS

In order to draw conclusions, we will evaluate each hypothesis, and then give a short summary of our research.

Hypothesis 1

In hypothesis 1 we tried to argue that heterogeneity of sex has a positive influence on the experienced level of creativity. We tried to test this hypothesis by considering the heterogeneity of research and development teams out of a given dataset. However, at single-level, the number of heterogeneous teams proved to be too small to determine a relation whatsoever. When we applied two-level regression both at team and at individual level, we found there was a positive relation between the heterogeneity of sex and the team creativity. Using this we can conclude mixing up teams sexually has a positive effect on experienced creativity.

Hypothesis 2

Hypothesis 2 argued that heterogeneity of age has a negative impact on the level of creativity. On first sight, when drawing a correlation figure, the results were not very promising. No clear relationship could be established. However, when the box plot figure was created of the data, it became clear that heterogeneity of age has a negative impact on the level of creativity. However, this relation is so weak, we found it was not significant by applying multilevel analysis.

Hypothesis 3

In the third hypothesis we claimed that heterogeneity of service years has a positive influence on the level of creativity.

To determine the relation between heterogeneity of service years and creativity in research and development teams we measured the heterogeneity of service years as a percentage. This percentage was then compared to experienced team creativity.

The result of this comparison first seemed to be undetermined, however, when a trend line was drawn we could determine a slight negative impact. This negative trend is totally the opposite of our hypothesis, which is an interesting result.

However, because the trend line is slight negative this relation is not very strong. To better test the hypothesis one should use a larger dataset. Also multilevel analysis showed is a very slight negative impact. However, this relation is not significant to conclude there is a relation between heterogeneity of service years, and creativity.

Hypothesis 4

In our last hypothesis, hypothesis 4, we claimed that diversity of education level has a positive influence on the level of creativity. On first sight, no correlation can be found in our results. However, we could find a concentration of teams at a creativity score of 30%. On both sides of this concentration there are a few results. To better analyze this data, we created a box plot figure from this data.

This box plot shows us that diversity of education level has a positive influence on the level of creativity, but when diversity is too high, it has a negative influence. Again multilevel analysis showed this relation is very weak, but not significant enough to conclude there is a relation between education level and creativity. This is also an interesting conclusion which might be researched further.

Summary

Recapitulating, we tried to find out if there is a relation between heterogeneity and experienced creativity of research and development project teams. To determine this relation we chose 4 general R&D team aspects *education, service years, sex* and *age* and measured their impact on experienced creativity.

Each of the aspects showed us different results on the experienced level of creativity. Only a relation between heterogeneity of sex and creativity could be proved. The other hypotheses showed us a very slight effect, but the result was not significant enough to conclude the is a relation. Using the first proven relation we could say heterogeneity of sex has an impact on the experienced creativity of R&D teams. This was not obvious at first sight, but better analysis of the data made this clear.

For the other hypotheses we suggest more research should be done.

Using the research results of the 4 attributes of heterogeneity, we conclude heterogeneity has a slight effect on the experienced creativity of a R&D team. Some attributes of heterogeneity have more effect on experienced creativity than others. But we depicted there is a relation.

The answer for the second main question follows from the conclusions stated above. The manager of an R&D team should compose a Research and Development team of people who are new in the organization.
Using a Research and Development team of mixed sex results in better experienced creativity. Also diversity in level of education seems to have a positive impact on the creativity. Using this information, the manager of such a team could balance the team to maximize team performance.

This research could be used in a various companies where there is a need for innovative, problem-solving R&D teams. Of course this does not guarantee improved creativity, but it should be used as a reminder.

Appendix A: Research Results

TEAM NR.	AGE (%)	SEX (%)	SERVICE YEARS (σ)	EDUCATION (%)	CREATIVITY
1	43	0	4.79	39	4.3
2	25	0	10.56	10	5.2
3	46	0	9.72	27	4.7
4	25	0	7.00	27	5.2
5	50	0	2.77	30	4.0
6	75	80	1.34	50	5.0
7	0	40	0.55	10	4.4
8	75	0	1.79	10	3.8
9	25	80	0.55	50	4.8
10	25	40	0.84	30	6.2
11	75	40	3.71	30	5.4
12	25	0	3.42	50	5.4
13	25	0	4.92	50	5.0
14	25	0	4.67	30	4.3
15	25	0	3.63	30	5.8
16	50	0	6.60	70	4.4
17	41	0	6.88	23	4.4
18	63	0	12.34	27	5.3
19	39	0	8.77	30	4.6
20	46	33	6.39	47	5.0
21	50	0	6.72	30	4.4
22	25	0	3.39	39	4.4
23	25	0	3.16	50	5.4
24	50	0	7.27	30	4.3
25	25	0	6.95	70	3.8
26	25	0	2.64	16	4.4
27	50	0	6.88	30	3.8
28	25	33	4.62	10	3.8
29	25	0	0.53	36	5.2
30	25	0	4.16	10	5.6
31	25	0	2.77	30	5.6
32	25	0	1.64	30	6.0
33	25	0	2.63	30	4.0

Table 9: Results

References

[1] Cady, S.H., and Valentine, J. Team innovation and perceptions of consideration. *Small Group Research*, 30(6):730–750, 1999.

[2] Chatman, J.A., & O'Reilly, C.A. Asymmetric reactions to work group sex diversity among men and women. *Academy of Management Journal*, 47:193–208, 2004.

[3] Kratzer, J. *Communication and Performance: an empirical study in innovation teams.* PhD thesis, Rijksuniversiteit Groningen, 2001.

[4] Pelled, L.H., Eisenhardt, K.M., and Xin, K.R. Exploring the black box: An analysis of work group diversity, conflict, and performance. *Administrative Science Quarterly*, 44:1–28, 1999.

[5] Schruijer, S.G.L., and Mostert, I. Creativity and sex composition: An experimental illustration. *European Journal of Work and Organizational Psychology*, 6(2):175–182, 1997.

[6] Timmerman, T.A. Racial diversity, age diversity, interdependence, and team performance. *Small Group Research*, 31(5):592–606, 2000.

[7] Wiersema, M.F., & Bantel, K.A. Top management team demography and corporate strategic change. *Academy of Management Journal*, 35:91–121, 1992.

[8] Zenger, T.R., & Lawrence, B.S. Organization demography: The differential effects of age and tenure distributions on technical communication. *Academy of Management Journal*, 32:353–376, 1989.

Level of Education, the Diversity of Field of Specialization, Problem-Solving Communication, and the Productivity of R&D Teams

J. Kizito • D. Tuheirwe
Rijks universiteit Groningen, Department of Mathematics and Computing Science
csg4048@wing.rug.nl • csg4049@wing.rug.nl

Acknowledgement
We would like to thank Dr. Jan Kratzer of the Faculteit Bedrijfskunde, Rijksuniversiteit Groningen for supporting our efforts with this paper. We would also like to thank the reviewers of our paper for their contribution.

Abstract
The productivity of Research and Development (R&D) teams is determined by a number of factors. This paper looks at a few of these factors namely highest degree as a measure of level of education, the diversity of field of specialization, and problem-solving communication. We further look at the effect of diversity of field of specialization on problem-solving communication.
The results showed that high degrees had a small positive impact on team productivity, diversity of field of specialization proved to have a negative impact on problem-solving communication, which, in turn, had a negative impact on team productivity. Finally, based on our research, we could not draw any conclusion regarding the effect of diversity of field of specialization on team productivity.

Introduction

Teams are the building blocks of many organizational structures. A team can often accomplish more than what its members could achieve when working independently. Gavish (1997) points out that tasks are frequently interdependent and thus, one task cannot be completed without the cooperation and coordination of other members within and outside of the organization. Gerard (1991) argues that teams are particularly good at combining talents and providing innovative solutions to possible unfamiliar problems. In cases where there is no well established approach/procedure, the wider skill and knowledge set of the team has a distinct advantage over that of the individual. The range of skills provided by a team's members and the self monitoring, which each team performs, makes it a reasonably safe recipient for delegated responsibility. Even if a problem could be decided by an individual, there are two main benefits in involving the actors in the decision process. Firstly, the motivational aspect of participating in the decision will clearly enhance its implementation. Secondly, there may well be factors, which the implementer understands better than the individual who could supposedly have decided alone.

To maintain a position in the market, organizations must continuously develop new products and business processes. A large body of research indicates that good technical communication within the R&D organization itself is essential for R&D productivity. As a consequence, the importance of communication networks in the R&D environment for successful innovation and new product development is already well acknowledged by both practitioners and researchers. More frequent contact and communication between team members improves coordination, such that task objectives are more likely to be realized.

Communication includes all interaction and information exchange between parties. Examples include verbal, written and electronic information exchange, such as the transmission of documents. In this paper we examine communication by studying the diversity of field of specialization and relating it to problem-solving communication and the effect of the latter on team productivity.

A team can be productive if its members are knowledgeable. Since one’s knowledge is a core product of education, team members with good education can greatly enhance productivity. In this research we examine this relation by studying the highest degree received by the team members and relating it with productivity. We consider the variation of degrees at team level and show how this affects the productivity of the team.

We also look at how diversity of field of specialization in a team can enhance productivity. Combination of different talents and the synergy it creates can overcome many difficulties encountered in organizational life, including production, planning and problem solving.

We discuss the relationship between the levels of education, diversity of fields of specialization, problem-solving communication and team productivity, and submit the hypotheses that we test in our analysis. Thereafter the methodological design of the research is described. Finally, results are presented and conclusions drawn from these results are discussed.

Theory

Impact of level of education on team productivity

Bynner et al., 2003 states that graduates are less depressed, healthier, more likely to vote in elections and help with their children's education. The advantages graduates derive from higher education cover not only better jobs and higher pay, but also a wide range of other personal and social benefits. Bynner et al., 2003 continues to state that research has it that students who dropped out of higher education before graduation showed a reduction in the indicators of good health compared with those who completed their studies and gained a degree. Higher education is a key driver in providing economic and social benefits in an organization in the sense that the knowledge, skills and attitudes of graduates enhance productivity when they solve problems together as a team.

H1: The higher the number of members with high degree qualifications in a team, the higher is the productivity.

Impact of diversity of field of specialization on team productivity

Ford and Randolph (1992) state that in a cross-functional structure, individuals have the opportunity to work on a variety of projects with a variety of individuals from across the organization. In sharing ideas, knowledge, and perspectives, the team enlarges an individual's experience and outlook, increases responsibility and involvement in decision making, and offers a greater opportunity to display capabilities and skills. Because greater demographic diversity entails relationships among people with different sets of contacts, skills, information, and experiences, heterogeneous teams enjoy an enhanced capacity for creative problem solving (Reagans and Zuckerman, 2001). The growing diversities of specialization in a team may be accompanied by a parallel increase in attitudinal or cognitive diversity of the team members. According to Kilduff et al., 2000, diversity in specialization signals diversity in underlying and invisible cognitive processes. From this perspective, diversity may have important effects on team and organizational performance. A heterogeneous team with members having diverse specializations can be more productive since the members will be knowledgeable in various areas.

H2: The more diverse the fields of specialization of the team members, the greater is the productivity.

On the other hand, diversity of specialization in a team creates an atmosphere of ambiguity and conflict as well as additional costs, both for the organization and for the individual. Ford and Randolph (1992) point out that the interaction of people with different work orientations (e.g., project/task vs. functional/professional), different professional affiliations, different time horizons (e.g., long term vs. short term), and different values are all potential causes of conflict. In a cross functional team, individuals find themselves working across various projects under different managers. This situation creates multiple reporting relationships (role conflict), conflicting and confusing expectations (role ambiguity), and excessive demands (role overload). Another major disadvantage is cost. Management can be costly for both the organization and the individuals in the organization. According to Ford and Randolph (1992), a team with diversity of specialization leads to costs associated with organizational "heaviness" including excessive meetings or "groupitis," which can lead to delayed decision making and increased information processing costs. The costs of unused or underused resources, both physical and human, are also likely to increase as well as the costs for extra training of project/matrix managers and the costs associated with monitoring, controlling, and coordinating the people and project within the team. All these mentioned disadvantages can lead to reduction in productivity.

Furthermore, homogeneous groups are expected to perform at a higher level because such groups coordinate their activities more easily than diverse teams, according to Reagans and Zuckerman (2001) based on the work of McCain et al., 1983, O'Reilly et al., 1989, Zenger and Lawrence 1989. It is further believed that these groups are more harmonious and communication between the team members is effective. Reagans and Zuckerman
(2001) recognize that diverse teams are likely to face significant difficulties because of a lessened capacity for coordination.

H3: The less diverse the fields of specialization of the team members, the greater is the productivity.

Impact of diversity of field of specialization on problem-solving communication

Differences in personality, training, background, departmental culture, and task priorities and responsibilities, result in strong language and attitudinal barriers between R&D and marketing professionals (Griffin and Hauser 1996). Similar barriers are often witnessed between marketing and operations management, and between R&D and operations. Such barriers imply that intra-functional communication will be more prevalent than cross functional communication (Christophe and Rudy 1977). Internal communication is influenced, e.g., by physical distance between team members (Allen, 1984), and the cohesiveness (Keller, 1986) and the homogeneity of a team (Ancona & Caldwell, 1992a; Bruce et al., 1995). Ancona and Caldwell (1992a) further studied the effects of team diversity on communication. They found that tenure homogeneity within a group increased the communication among team members.

H4: The less diverse the field of specialization of team members, the higher is the frequency of problem-solving communication among the members.

Impact of problem-solving communication on team productivity

Good communication is all you need to run a highly competitive, successful business. In fact good communication is the only successful way to run a knowledge-based business (Herrington 2004). Most studies find that increased internal and external communication affect a project's performance positively (e.g. Allen, 1984). The empirical findings of Pelz and Andrews (1966) coupled with the longitudinal studies of Allen (1970) and Farris (1969) strongly support the contention that direct communications between project group members and other internal professional colleagues can enhance project effectiveness.

H5: The higher the frequency of problem-solving communication, the greater is the team productivity.

Method

We exploit survey data on 199 team members in 33 innovation teams, which was gathered in 11 Dutch companies that are conducting innovation activities. All 11 companies are engaged in production and innovation of digital products. The data were collected using questionnaires distributed and filled out during team meetings. Due to this method the response rate was very high with 95 percent.

Main dependent variables

Team Productivity: This is a measure of how productive the team, in the sense of producing information, devices, materials, etc. to develop a prototype into a fully-fledged product is. According to the underlying meaning of the variable 'team productivity', this variable was measured by asking the team members to rate themselves on a 7-point scale [from 1 (much worse) to 7 (much better)]. This was transformed to team level by computing the average (or mean) of the individual values.

Problem-Solving Communication: This is a measure of how often team members talk to one another concerning the discussion, development, or evaluation of new ideas or approaches to technical problems, technical or scientific help or advise and/or the distribution of scientific or technical information (stemming from in and/or outside the company). For each pair of members in a team, this variable was measured using the following possible values:

1. Never
2. Less than once a month
3. 1 to 3 times a month
4. 1 to 3 times a week
5. Once daily
6. More than once a day

This variable was transformed to team level by computing the average of the individual mean values.

Main independent variables

Highest Degree: This is the highest degree received by a member with the following possible values: □

1. Elementary Education
2. Secondary Education
3. Higher Secondary Education
4. Polytechnic Education
5. Academic Education / University
6. PhD

This was transformed to team level by computing the median of the individual values.

Field of Specialization: This is a measure of the area that best represents a member's major field of specialisation. This variable was measured using the following possible values: □

1. Biological Science
2. Business Administration / Economics
3. Chemistry / Chemical Engineering
4. Electrical Engineering
5. Mechanical Engineering
6. Mathematics, Statistics or Computer Science
7. Medical Sciences
8. Physics / Physical Engineering
9. Social Sciences
10. Others

The diversity of field of specialization was transformed to team level by computing the paired difference index of the individual values.

A summary of these computations is given in table 1.

Insert Table 1 about here

In some of these computations, there was missing data. If a team member's value was found missing, he/ she was left out of the computation. If data for the entire team was missing, the team was ignored.

Analysis

For the analysis of data we used multivariate regression. Let the function Y denote the dependent variable. The general form of multiple regression models is

$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \varepsilon$

The dependent variable Y is written as a function of k independent variables x_1, x_2, …, x_k. ε is a random error term added to make the model probabilistic rather than deterministic. We assume that for any given set of values of x_1, x_2, …, x_k, the error term has a normal probability distribution with mean equal to 0 and variance equal to σ^2. The error associated with any one Y value is independent of the error associated with any other Y value. The value of the coefficient β_i determines the contribution of the independent variable x_i, and β_0 is the Y intercept (McClave et al., 2005). For this case,

Y = Team Productivity
x_1 = Highest Degree
x_2 = Field of Specialization
x_3 = Problem□Solving Communication

For the analysis of the relationship between field of specialization and problem□solving communication, we used a bivariate correlation.

Results

Table 2 presents the multivariate regression coefficients regarding the team perspectives.

Insert Table 2 about here

The regression equation now becomes *Team Productivity* = 4.584 + .232 *Highest Degree* + .164 *Field of Specialization* + □ .346 *Problem-Solving Communication*

The intercept (4.584) does not have a meaningful interpretation since setting the values of all the independent variables to 0 is not practical.

β_1 (.232) shows that team productivity increases by .232 for every unit increase in highest degree when both field of specialization and problem-solving communication are held fixed.

β_2 (.164) shows that team productivity increases by .164 for every unit increase in field of specialization when both highest degree and problem-solving communication are held fixed.

β_3 (−.346) shows that team productivity decreases by .346 for every unit increase in problem-solving communication when both highest degree and field of specialization are held fixed.

Table 3 shows that the correlation regarding field of specialization and problem-solving communication is statistically significant (−.275). This means that the less diverse the field of specialization of team members, the higher is the problem solving communication among the members.

Insert Table 3 about here

In summarizing the results it can be stated that highest degree has a positive impact on team productivity. Diversity of field of specialization does not seem to have an effect and problem-solving communication has a negative impact on team productivity. Sometimes an increased intensity of communication is associated with increased productivity and improved performance. Whereas sometimes, there is no relationship and other times there is a negative relationship (Kratzer 2000). Furthermore, diversity of field of specialization has negative impact on problem-solving communication.

According to the results, hypotheses 1 and 4 can be partly confirmed. Hypotheses 2 and 3 cannot be confirmed since the relationship between diversity of field of specialization and team productivity is insignificant. We reject hypothesis 5.

Discussion and Conclusion

In this research, we investigated the effect of highest degree, diversity of field of specialization, and problem-solving communication on team productivity. We also found out the relationship between the diversity of field of specialization and problem-solving communication.

In managerial terms, the results imply a number of things about the composition of R&D teams. Members with high degree qualifications are somewhat more productive. This could be because there are few teams having members with high degrees that are more productive and there are more teams, which are less productive, having members with lower degrees. Therefore managers should ensure that there is a significant number of members with high degrees.

We realize that if people of similar skills work together, problem-solving communication is enhanced. However, if this communication is too much, it has a negative impact on team productivity. Managers should thus keep it at an optimal level.

We were unable to confirm the relationship between diversity of field of specialization and productivity. There was a positive, though insignificant, relationship. Future research on this variable may be carried out to further investigate the relationship.

References

Allen, T.J. (1970) Communications networks in R&D labs. R&D Management, 1, 14□21.

Allen, T.J. (1984). Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information within the R&D Organization. Cambridge: The MIT Press.

Ancona, D. & Caldwell, D. (1992a). Demography and Design: Predictors of New Product Team Performance. Organization Science, Vol. 3, No. 3, August.

Bruce, M., Leverick, F., Litter, D. & Wilson, D. (1995). Success factors for collaborative product development: A study of suppliers of information and communication technology. R&D Management, 25, 1, 33□44.

Bynner, J., Dolton, P., Feinstein, F., Makepeace, G., Malmberg, L., Woods, L. (2003). Benefits of higher education reach far beyond the job market.
Retrieved October 2004, from, http://www.hefce.ac.uk/News/HEFCE/2003/benefit.htm.

Christophe, B. (The Pennsylvania State University) and Rudy, K.M. (Free University of
Brussels), "The Effects of R&D Team Co□location on Communication Patterns Among
R&D, Marketing, and Manufacturing," ISBM Report 7□1997.

Farris, G. (1969). Organizational factors and individual performance. Journal of Applied
Psychology, 53, 86□92.

Ford, C.R., Randolph, A.W (1992). Cross□functional structures: a review and integration of matrix organization and project management.
Retrieved October 2004, from, http://www.findarticles.com/p/articles/mi_m4256/is_n2_v18/ai_12720959/pg_4

Gavish, B. (1997). The Impact of Information Technology on the Organization of Teams.
Retrieved October 2004, from, http://www.beje.decon.ufpe.br/article1.htm

Gerard, B. M. (1991). Groups That Work.
Retrieved October 2004, from, http://www.ee.ed.ac.uk/~gerard/Management/art0.html

Griffin, A. and J.R. Hauser (1996). "Integrating R&D and Marketing: A Review and
Analysis of the Literature," Journal of Product Innovation Management, 13, 191□215.

Herrington, A. (2004). Communicating is business effective management.
Retrieved November 2004, from, http://www.pateo.com/abtpat1.html

Keller, R.T. (1986). "Predictors of the Performance of Project Groups in R&D Organizations," Academy of Management Journal, 29, 7 15□26.

Kilduff, M., Angelmar, R., & Mehra, A. 2000. Top management team diversity and firm performance: Examining the role of cognitions. Organization Science.

Kratzer, J. (2000). Communication and performance: An Empirical Study in Innovation
Teams, Thesis Publisher: Amsterdam (Ph.D).

McCain, Bruce E., Charles O'Reilly, Jeffrey Pfeffer. (1983). The effects of departmental demography on turnover: The case of a university. Acad. Management J. 26 626□641.
McClave, T.J., Benson, P.G., Sincich, T. (2005). Statistics for Business and Economics.
London, Pearson Education Inc.

O'Reilly, Charles A. III, David F. Caldwell, William P. Barnett. (1989). Work group demography, social integration, and turnover. Admin. Sci. Quart. 34 21□37.

Pelz, D. & Andrews, F. (1966). Scientists in Organizations. New York: Wiley.

Reagans, R., Zuckerman, E.W. (2001). Networks, Diversity, and Productivity: The Social
Capital of Corporate R&D Teams. Vol. 12, No 4, July□August 2001.

Zenger, Todd R., Barbara S. Lawrence. (1989). Organizational demography: The differential effects of age and tenure distributions on technical communication. Acad.
Management J. 32 353□376.

Table 1: Descriptive statistics of team level variables

Variable	Mean	Median	Variance	Minimum	Maximum	No. of Teams
Highest Degree	□	4.000	□	2.0	6.0	33
Field of Specialization	□	□	0.075	0.0	0.9	33
Problem□Solving Communication	3.186	□	□	1.7	5.0	33
Team Productivity	4.544	□	□	3.4	6.4	31

Table 2: Multivariate regression coefficients for highest degree, field of specialization, problem□solving communication, and team productivity

		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
Model		B	Std. Error	Beta		
1	(Constant)	4.584	1.303		3.517	.002
	Highest Degree	.232	.200	.208	1.156	.258
	Field of Specialization	.164	.568	.052	.288	.776
	Communication	□.346	.230	□.276	□1.504	.144

Table 3: Bivariate correlation for field of specialization and problem□solving Communication

			Communication
Spearman's rho	Field of Specialization	Correlation Coefficient	□.275
		Sig. (2□tailed)	.121
		N	33