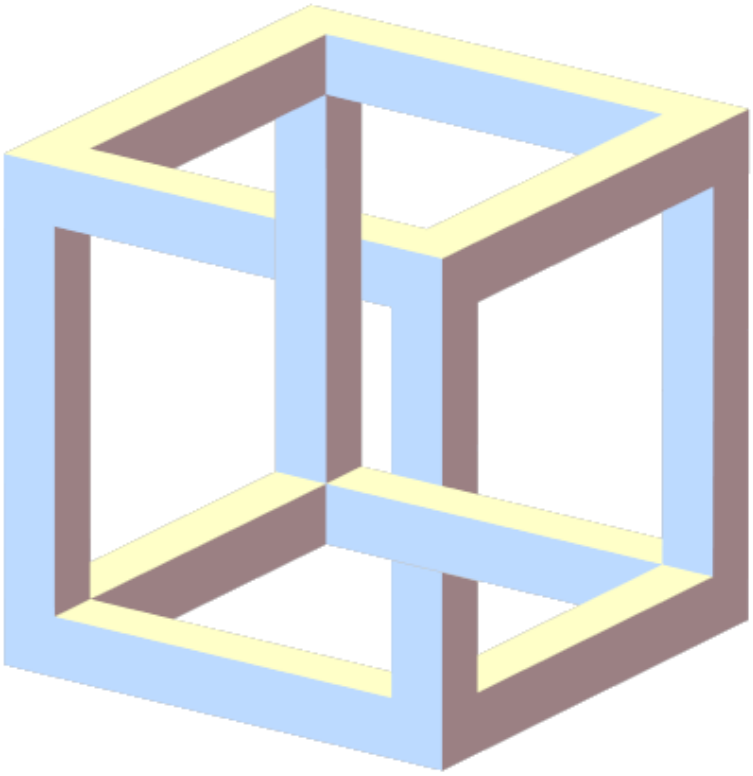
3^{rd} Year Project Handbook 2019-2020

By Markus Roggenbach, Liam O'Reilly and Ben Mora

As of September 2019



Dear students!

Here it is, our 3^{rd} Year Project Handbook, meanwhile in its 12^{th} edition! This handbook shall guide you throughout your entire project: It tells you

- which deliverables the department requires from you, namely
  - Initial Document,
  - Presentation in Gregynog,
  - Dissertation, and
  - Stand at the Project Demonstration Fair,
- what topics you should address in each of them,
- when they are due, and
- how they are marked.

The project is a major part of your studies. It counts for 30 credits of your final year (i.e one quarter of your marks). This is a good reason for putting some effort into it – and of course it should be a lot of fun. It is actually your project, you are in the driving seat, you will take all major decisions!

Besides this deliverable-oriented view, this handbook also gives you background information that might come handy. Nearly all of you will deliver a Software Product. Thus, some reminders on the topic of Software Engineering appear to be appropriate – particularly those topics that are important to the project. You will write three documents – a chapter on document writing gives you some hints about this. You will give a presentation on your project – there is a chapter on presentation techniques. You will exhibit the results of your project at the Project Demonstration Fair – yet another chapter discusses how to come up with a fancy stand.

As projects come in different flavours any Handbook will fall short to give fully matching advice for all of them. Therefore, students and supervisors are encouraged to reflect to what extent the recommendations of this handbook apply to the specific settings of their project.

Adapting what you deliver for your project can be illustrated in terms of an example. Many projects will need a User Handbook: but not all (and some will require a more complex one than others). Consider three different types of project.

1. In a project, say, on experimental implementations on "Exact Real Number Computations", a User Handbook will be of no importance at all. This insight, however, can be documented, e.g., in the project plan: "Deviating from the standard of our development model, we omit an explicit User Handbook. As our software is command line based, it will be sufficient to compile a short table summarising the small number of different calls possible.' The Dissertation should repeat the rationale why there is no comprehensive User Handbook and include the (perhaps updated) table replacing the User Handbook in the appendix.

2. A project that develops a tool on "Test Coverage", where the user uploads a program, the user enters some test cases, the user selects a coverage criterion, and the tool computes if the given test suite covers the program according to a selected criterion – see its GUI in Figure 1 – will naturally require a User Handbook. A short User Handbook will be appropriate thanks to the simple nature of the interaction between user and program.

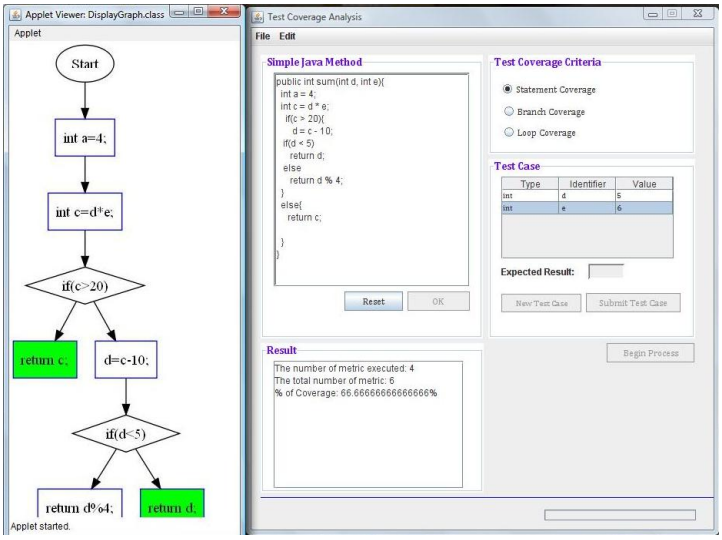


Figure 1: GUI of a Testing Tool developed by K Ayodeji in his student project.

3. A project on "Student-Project Administration", finally, will have to include a comprehensive User Handbook: The software shall support the administration of Student Projects in their various phases – the selection phase, the allocation phase, the submission phase, the marking phase, just to name a few. Additionally there are different roles: the administrator, the students, and the lecturing staff. During the various phases users in different roles can perform different interactions with the system. This needs careful documentation in an adequate form. In the end, the different users shall be able to successfully use the program based solely on the information given in the User Handbook.

This said, let's hope that the handbook helps, and that thanks to it no final year project falls short from being fantastic. I at least tried my best. Have fun reading my pamphlet, and all the best for your studies, especially for your project!

Contents

1 Final Year Project timeline and modules

The final year project has a start up phase in **year 2**. In this phase students are mainly concerned with project selection – however, they are encouraged to start on their projects over the summer between Level 2 and Level 3.

before Easter	Project Selection Brochure handed out
after Easter	Project Selection Fair
last week of 2nd semester	Visit Project Demonstration Fair
end of 2nd semester	Project Selection
end of May/June exams	Project Allocation
summer	First meetings with supervisor & background reading

The main body of work then takes place in **year 3** - it is structured in terms of 'deliverables':

Initial document	0?/11/2019
After the January assessment period	
Gregynog Presentations	24/11 to 30/11 2019
Before the Easter break	Final Dissertation Outline (recommended only)
After the Easter break	Final Dissertation
Before the exams	Project Demonstration Fair

In total the final year project is worth 30 credits:

- Its development is overall worth 15 credits. The development module comprises of the Initial Document, the presentation at Gregynog, and the presentation at the Project Demonstration Fair.
- The project Dissertation is worth 15 credits.

Naturally, there will be an overlap between the Dissertation and the Initial Document. These documents address questions such as motivation of the project, background studies, technology involved, first prototype, management etc. The discussion of these topics in the Initial Document however, will be premature, as these documents have been produced in early project phases.

Thus, while parts of the project Dissertation can be based on parts of the Initial document, it is expected that these parts are re-viewed and altered in the light of the deeper insight gained in the final stages of your project.

This said, one should also point out that the dissertation:

- should be a self-contained document
- should clearly demonstrate progress beyond the results already mentioned in the earlier documents.

Finally, due to the large cohort of students, we have decided to use the current supervision policy:

- At a minimum, we will hold 2 compulsory academic mentoring sessions each semester:
- The first one must be a half-hour an individual meeting per student where Academic Mentoring takes place and also discussion of the project.

  -The other can be a group meeting.

- In addition supervisors will allow students to book meetings to discuss their projects:

  -Each student can have up to 3 hours each semester of such meetings. Students must contact supervisors to organise these.

  -It is up to supervisors to decide how this works. E.g., they could be group meetings or individual meetings.

It must be added that any concerns regarding supervision or more generally any issues regarding the processes in place to manage third year projects can be reported to the year head(s) (preferred option) or director of teaching.

2 Project flavours

3^{rd} year projects come in different "flavours". There are

- Computer Science "Non-Technical" Projects:
  CSP354 – Computer Science Project Specification and Development
  CSP302 – Computer Science Project Dissertation

- Computer Science "Technical" Projects:
  CSP354 – Computer Science Project Specification and Development
  CSP344 – Computer Science Project Implementation and Dissertation

- Software Engineering Projects:
  CSP301 – Software Engineering Project Specification and Development
  CSP300 – Software Engineering Project Implementation and Dissertation

The time-line presented in Section 1 applies to all flavours, i.e., all students will submit an Initial Document and a Dissertation, and all students will give presentations in Gregynog and at the Project Demonstration Fair – where the same deadlines apply to everyone. The main difference between the projects is what one might want to call the nature of the project management component.

For Computer Science "Non-Technical" Projects, you might want to consider research methodologies for the humanities, when you plan your project. These are, e.g., discussed in

Willie van Peer, Frank Hakemulder and Sonia Zyngier:
Scientific Methods for the Humanities.
John Benjamins Publishing Company, 2012.

Computer Science "Technical" Projects, need to include references to a chosen software-lifecyle (SLC) model. Reflecting their scientific nature, additionally, they can also include references to a research methodology in computer science, see e.g.

About Computing Science Research Methodology
available at:
`https://webdocs.cs.ualberta.ca/~c603/readings/research-methods.pdf`

Finally, **Software Engineering Projects**, need to choose a Software Life Cyle (SLC) model, and say how this shall be applied in a light-weight manner throughout the project. In their Dissertation they will have to include an appendix comprising of the deliverables according to their chosen light-weight application of their SLC.

It is recommended that students clearly state the flavour of their project in the beginning of all their project documents.

3 Initial Document

At the beginning of week 5 of the first semester in year 3 the students hand in an "Initial Document" on their projects. It is worth 40% of the final mark.

3.1 What it is about

In this Initial Document (about 12-18 core pages + appendix if needed) the students shall demonstrate that they

- understand the motivation of their project,
- have a clear project aim,
- know about related work,
- understand the topics revolving around their project,
- have done some requirement analysis and have chosen a software development model (if appropriate/SE students) – see Section 8.1
- have a realistic schedule for the work involved – that reflects the choice of model, see 8.2 – and
- are aware of potential risks – see Section 8.3.

3.2 Hints for writing

The students shall discuss structure and contents specific to their Initial Document with their supervisor. General rules for writing are outlined in chapter 9 "Writing documents" of this handbook.

3.3 Submission

By the deadline (0?/11/2019) students are asked submit an electronic copy (as a .pdf file) to the Blackboard Turnitin site. In case of late submission the standard university rules apply. You may also be asked to submit an extra copy by email but we will notify you first if this is needed.

3.4 Marking

The Initial Document is worth 40% of the mark in the development module. It is assessed by the project's supervisor & a chosen second marker using the marking scheme available at:
https://cs.swansea.ac.uk/~csmora/project/Marking/2019-2020/MarkingForms.php

4 Presentation in Gregynog

At some point during the (third) year, the Computer Science Department will organise its annual Scientific Undergraduate Conference in Gregynog – see Figure 2. At this conference the students give a presentation regarding their project.

Figure 2: Student levitation at Gregygnog 2010.

4.1 What it is about

In this presentation (10 to 15 minutes for the presentation + 3 to 5 minutes for discussion – the duration of the time slots is determined newly every year by the respective conference organisers; there will be a notification in due course) the students shall convince the audience – fellow students as well as lecturing staff – that they

- have an exciting topic for their final year project,
- are on top of tools & techniques involved, and
- have made good progress on their project so far (i.e. dates: 24/11 to 30/11 2019).

The basic set-up is that students & staffs are allocated to streams (about 15 students and 2 staff per stream). Students give their talk within their allocated stream and listen to the talks of the students of that stream. There will be a computer and a data projector available in order to present slides. The computer will support the pdf-format.

Note: For any other format used, any tools that should be demonstrated, it is the students' responsibility to ensure that they can be shown, e.g., by bringing their own laptop and ensuring that it works with the data projector available in the conference room allocated to the stream.

4.2 Hints for preparing the presentation

The students shall discuss structure and contents specific to the presentation with their supervisor. General rules are outlined in chapter 10 "Making presentations" of this handbook.

Dissertation Outline

1. **Introduction** Increasing software complexity requires more sophisticated testing methods – Data Protection Act prevents use of "real data": 1st reason for data generation – have "good" test data: 2nd reason for data generation – Industrial Partner: Grid Tools – Project aims: generation of data with trends, demonstrated on Credit Card Fraud, integrated in DataMaker.

2. **Industrial Relation** *Company Grid Tools – Visits and cooperation –Tool DataMaker: What it is for, how it works, what it is programmed in, what its interfaces are – What we know about Credit Card Fraud detection*

3. **Testing** *Testing in general, with focus on black-box testing – distinction between test case and test configuration – Decision Tables in testing*

4. **Genetic algorithms** *The basic idea – Crossover, mutation, evaluation – . . .*

etc.

Figure 3: A sample outline

4.3 Marking

The presentation in Gregynog is worth 30% of the mark in the development module. It is assessed by two members of lecturing staff using the marking scheme available at:

https://cs.swansea.ac.uk/~csmora/project/Marking/2019-2020/MarkingForms.php

We will also notify you to submit the slides you presented to blackboard, possibly once you are back in Swansea. This is solely for the purpose of making the material available to our external examiner. In the unlikely case your presentation does not involve electronic slides, you may still be required to submit some material.

5 Dissertation outline (voluntary)

Before the Easter break, students are recommended to hand in a dissertation outline to their supervisors, see Figure 3 for an example. The outline shall cover the topics suggested in Section 6.1. This outline is to be discussed with the supervisor. Based on this outline, student and supervisor agree on organisation, content, and depth of the thesis. The dissertation outline is not assessed.

6 Dissertation

After the Easter break of year 3 the students hand in a dissertation on their project (likely to be electronic only submission). We would like to emphasize that you may find yourself very busy

around this time of the year so don't be late and don't start working on your document at the last minute. If you have managed well your project, then this should not be a problem.

6.1 What it is about

This dissertation is a comprehensive document (we will enforce this year **40 core pages** maximum; extra material can be located a section of the appendix) in which the students compile their project results. 40 – 50% of the dissertation shall be on the project background. The other 50 – 60% of the dissertation shall discuss the student's own results.

Depending on the nature of the project, the dissertation will take different forms. For Software Engineering Projects, e.g., the dissertation can address the following topics:

- The main technical problem(s) that have been solved.
- A brief survey on similar work.
- A study and survey of relevant literature.
- A table or tables linking statements made in the dissertation to the evidence provided in the appendix.
- Any consideration for the legal, social, professional and ethical issues related to the system if any (or explicitly addressing why they do not apply).
- Discussion of tools that were used.
- Reflection on the project results (e.g. reliability, functionality, compliance with specification).
- Suggestions for further work.

Students following a Software Engineering curriculum will also have to discuss:

- A description of the chosen software life-cycle model (SLC).
- A clear and precise narrative description of the project supported by references to the deliverables (requirements, design, implementation, testing, user manual, etc.) as prescribed by the initial document and appropriate for the chosen SLC.
- For Software Engineering Projects, the dissertation needs to include an appendix (not included in the above page count) which contains all deliverables as prescribed by the initial document and appropriate for the chosen SLC.
- Additional elements of Software Engineering, including for instance User Requirements, Specifications, Quality, Test plan, Software Designs (e.g., class diagram), etc.. Some of these elements may not apply, but you may want for instance to show a good program design for your project if programming is indeed an essential project activity.

This list is for guidance and is not a checklist – not all sections will be appropriate in all cases, and in some cases other topics not listed should be included.

6.2 Hints for writing

The students shall discuss structure and contents specific to the dissertation with their supervisor. General rules are outlined in chapter 9 "Writing documents" of this handbook.

Note that the department has a prescribed title page format – see Appendix B.1. The dissertation also needs to include the signed declarations and statements as shown in Appendix B.2.

Word and Latex templates of the title page and declarations will be made available to you as well.

6.3 Submission

You will submit your dissertation electronically through blackboard turnitin. Any code should be sent to our supervisor as well and we may ask you to also submit your project files electronically. Further emails before the submission deadline will let you know about how to proceed.

6.4 Marking

The dissertation is worth 100% of the dissertation module. It is assessed by the supervisor and the second marker of the project using marking schemes shown in Appendix and available on the net at:

https://cs.swansea.ac.uk/~csmora/project/Marking/2019-2020/MarkingForms.php.

For Software Engineering Projects, the appendix of deliverables is marked by sampling.

7 Project Demonstration Fair

Before the exams of the 2nd semester in year 3 the students present their project to the departmental public as well as to the local IT industry at the Project Demonstration Fair – see Figure 4.

7.1 What it is about

The fair offers the students a professional environment consisting of an exhibition stand (a panel, ca. 0.8 m high, 1m wide, usually with a laptop-table and one power socket) The students shall use the stand

- To draw attention to their project
- To provide the relevant background information
  (What is the project about? – What methods were used? – How was the problem solved? – What are the results?)
- As basis for shorter (maybe about 2 minutes) and longer (maybe about 6 minutes) presentations to visitors to their stand;
- A demonstration of the running software if appropriate.

Special requirements The fair organisers will provide a standard set-up for all students as described above. If a student's project requires a special set-up, say an internet connection, it is the student's job to organise this in co-operation with the fair organisers.

Figure 4: Project Demonstration Fair, 2008.

7.2 Hints for designing a stand

The students shall discuss structure and contents specific to their project with their respective supervisor. General rules are outlined in chapter 11 “Designing posters” of this handbook.

7.3 Submission

On the day before the Fair, the student submits an electronic copy of their poster(s) (as .pdf or .doc) to the Blackboard Turnitin site. In case of late submission the standard university rules apply.

7.4 Marking

The presentation at the fair is worth 30% of the mark in the development module. It is assessed by two members of lecturing staff using the marking scheme available at: https://cs.swansea.ac.uk/~csmora/project/M 2020/MarkingForms.php. Only the presentations to these two member of staff are marked. Ideally, these members of staff are the supervisor and the 2nd marker.

8 Software Engineering

The department offers various modules covering the subject of Software Engineering and all project students will have studied the subject. The following chapters are intended to remind students of

key concepts, and to give guidance specific to the 3^{rd} Year Project. Standard literature on Software Engineering such as

- R E Fairley: *Software Engineering Concepts*, McGraw-Hill, 1985.
- I Summerville: *Software Engineering.* 8th Edition, Prentice Hall, 2007.

(both of which are recommended reading for the Department's Software Engineering modules) will provide material for background reading.

8.1 Software Development Models

The students shall choose a Software Development Model appropriate to their project. In the Initial Document as well as in the final Dissertation, the students shall argue

- why the model was chosen initially.
- Whether this choice was a good one.

To this end, the students shall discuss the pros and cons of the chosen model. It also might be helpful to discuss alternatives.

The actual Software Development Model chosen will depend very much on the actual project. In an Industrial related project, just to take an example, often the *Prototyping Model* is chosen. Its slogan is:

Plan to throw one away, you will anyhow!

The Prototyping Model

- Is suitable for a system, where it is impossible to give an adequate system specification without a first, exploratory development or some experimentation;
- Yields fast a working system (with restrictions!) – this can be a good motivation for the student, and also strengthen the bond with the Industrial partner.

Prototyping is only one possible model – other possibilities include the "traditional" Waterfall Model or some kind of *Agile* model. Students should choose a model that is appropriate to their project, and explain why they have chosen it in their Initial Document.

8.2 Milestones and deliverables

Software Development Models prescribe development phases. Usually, at the end of each such phase a number of deliverables is due: programs, specifications, documents. In their role of project planners, the students shall make an appropriate selection of these deliverables and fix times when they are due.

For instance, when applying Extreme Programming, one would expect (a subset of) the following for each deployment:

- Progress review
- Selection of stories

- Breakdown of the stories into tasks
- Testing: methods, test cases, documentation of test results (also for regression test)

Here, it might be appropriate to adapt the deployment rate from once a week to once every fortnight.

In their role as project managers the student check

- Whether the deliverables are there in time and
- Whether their content / the results established allow the project to proceed to the next phase.

In case that the project is late (the usual case) or the established results are not as expected (also quite often the case), the student shall adjust the project plan accordingly: this can and will often include an adjustment of the initially stated project aims.

Such change of plan, if well-grounded and not due to bad planning in the first place, will be considered positive in the marking: It demonstrates good project management.

For the Final Year Project the students are asked to adapt a software development model to the project's needs. They will define a set of milestones as well as the deliverables due at each milestone. Naturally, the students will include (part of) these deliverables and into their Dissertation.

8.3 Risks in Software Development

Each project faces *risks* that affect its chances of success. Some risks are serious and some are not; some are likely to occur and some are not; and some are general to any project, while others will be specific. Here is a short list summarising typical risks that may occur in a 3^{rd} Year Project and how one might manage them:

- *Unrealistic time plan & unrealistic budget.* This is of course extremely common in many kinds of project. **Action:** More detailed planning with different cost estimation approaches; incremental development; re-use of software; reduction of the requirements if it becomes clear the original ones cannot be met.
- *Developing a product with the wrong functionality.* While it is unlikely that a project would be completely wrong, it is common for *parts* of a project not to be useful for their intended purpose because of a failure to understand precisely what was required. **Action:** User studies; analysis of weaknesses; early User Handbook; analysis of the information flow; prototyping.
- *Developing a product with an inadequate User Interface.* It is common for developers not to consider that (a) users of their software are not experts in what it does and how it works, and hence make mistakes; (b) not everyone uses software in the same way. So interfaces are often too rigid, and not intuitive or logical to everybody. **Action:** Use cases; user studies; characterising of different users; prototyping.
- *"Gold plating" of the product.* Commonly, the project as described in the Initial Document is too ambitious. **Action:** Reduction of the Requirement Document; prototyping; cost adequate design; cost analysis.
- *Permanent change of requirements.* This is not common in most projects, but can *sometimes* occur in Industrially-Related projects. **Action:** Choice of an incremental development model; choice of a design which is generic and easy to maintain.

- *Mistakes in third party components (e.g. libraries).* Although not common, this can and does occur. **Action:** Tests; inspections; compatibility analysis.
- *Real-time / Load problems.* Particularly in the early phases, testing tends to focus on "simple" cases that don't stress an application. **Action:** Simulations; load tests; modelling; prototyping; tuning.
- *Lack of background knowledge.* This is often the case when students fail to adequately research their project over the Summer Vacation between Level 2 and Level 3! **Action:** Background reading; choice of modules in the 3^{rd} year.
- *Wrong internal interfaces.* This occurs more commonly than would be expected, given that – in the setting of a 3^{rd} Year Project – the same person has written both "sides" of the interfaces. **Action:** Careful design and its validation; incremental development of operational sub-systems; prototyping.

In their Initial Document students shall identify potential risks of the project – maybe risks listed above, maybe risks far more specific to the very nature of their projects. In the dissertation students shall reflect on risks and their management during the project in the chapter on Evaluation.

8.4 Software Architecture

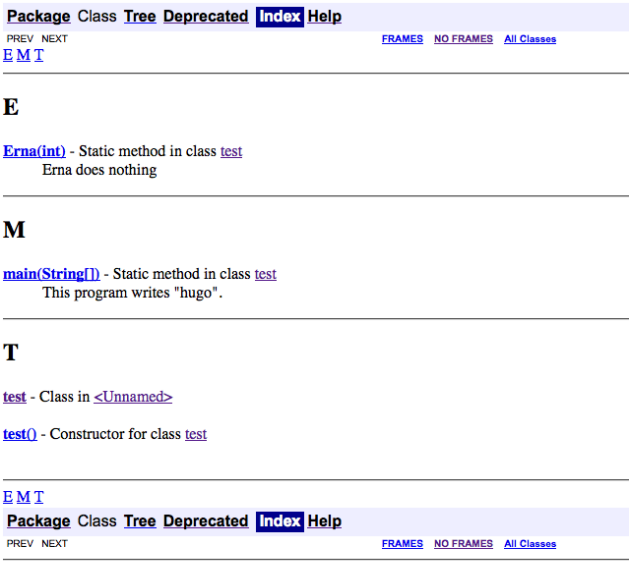


Figure 5: A documentation sample produced with Javadoc.

The design phase of any software development model results in a description of the system architecture to be implemented. The dissertation should include a description of the Software Architecture. Naturally, this description will be more detailed in the Dissertation.

Students should include part of the code documentation in the appendix of their Dissertation (or already into the previous documents if appropriate). The CD with the software should have a directory comprising of the full documentation.

Making the software architecture accessible Documentation generators such as Javadoc from Sun Microsystems provide practical means for making the software architecture accessible. Javadoc automatically generates documentation in HTML format from Java source code. Figure 5 gives an – admittedly meaningless – example of such a documentation.

The "doc comments" format used by Javadoc is the de facto industry standard for documenting Java classes. They can also been used in order to provide part of the code comments required – see Section 8.5. Tools similar to Javadoc exists also for other languages than Java, e.g., Haddock (`http://www.haskell.org/haddock/`) does the job for Haskel.

8.5 Maintainable code

During their projects, most students will produce loads of program code. This code should have a more comprehensible form than:

```
#!/usr/bin/perl -s
## Ian Goldberg <ian@cypherpunks.ca>, 19980817
$f=$d?-1:1;4D=pack(C',33..86);$p=shift;
$p=~y/a-z/A-Z/;$U='$D=~s/.*)U$/U$1/;
$D=~s/U(.)/$1U/;';($V=$U)=~s/U/V/g;
$p=~s/[A_Z]/$k=ord($&)-64,&e/eg;$k=0;
while(<>){y/a-z/A-Z/;y/A-Z//dc;$o.=$_}$o='X'
while length($o)%5&&!$d;
$o=~s/./chr(($f*&e+ord($&)-13)%26+65/eg;
$o=~s/X*$//if $d;$o=~~s/.{5}/$& /g;
print$o/n;sub v{$v=ord(substr($D,$_[0]))-32;
$v>53?53:$v}
sub w{$D=~s/(.{$_[0]})(.*)(.)/$2$1$3/}
sub e{eval$U$V$V;$D=~s/(.*)([UV].*[UV])(.*]/$3$2$1/;
&w(&v(53));$k?(&w($k)):($c=&v(&v(0)),$c>52?&e:$c)}
```

More concretely: Students are required

1. to comment their code, and
2. to format their code.

This holds especially for those code sample which are presented in the various documents, in the Gregynog presentation, or at the Project Demonstration Fair.

Commenting There are many code commenting standards, none of which is prescribed by the department. However, students *are* expected to choose (and use) one. Below is a brief, naturally incomplete compilation of a few such standards:

- Robert S Laramee: *A Source Code Comment Standard*

  `http://www.cs.swan.ac.uk/~csbob/teaching/laramee07commentConvention.pdf`

- *13 Tips to Comment Your Code*

Rule: Variable declaration

Explain for each variable
what it represents
in the context of the program.

Example:

```
[] int p     /* polynomial */
int    i,j   /* counters   */
```

Figure 6: A sample rule on code commenting in Java.

`http://www.devtopics.com/13-tips-to-comment-your-code/`

- Bernhard Spuida: *The fine Art of Commenting*

  `http://www.icsharpcode.net/TechNotes/Commenting20020413.pdf`

Comments are usually classified into documentary comments and comments along the program structure; students should do both. Documentary comments usually include:

1. Filename,
2. Version number/build number,
3. Creation date,
4. Last modification date,
5. Author's name,
6. Copyright notice,
7. Purpose of the program, and
8. Version history.

Commenting along the program structure requires one to attach comments to

1. variable declarations,
2. branching structures,
3. loops, and
4. methods.

Figure 6 shows a typical rule on how to comment a variable declaration.

Code formatting Many companies define an own style of how to format program code. Below are some sample links on such style guides:

- JavaRanch – a friendly place for Java greenhorns
  `http://www.javaranch.com/style.jsp`
- GeoSoft – Geotechnical Software Services
  `http://geosoft.no/development/javastyle.html`

In the context of Java Programming, the Sun Code Conventions can be found online by googling "java sun code convention pdf". This provides a de facto standard. Here a sample concerning the number of declarations per line:

```
int level;      // indentation
int size; // size of table
```

is preferred over

```
int level, size;
```

Tools like Checkstyle

`http://checkstyle.sourceforge.net/`

help programmers to write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task.

8.6 Testing

Testing means to systematically experiment with a system in order to establish a quality of this system. In the perception of the Software Engineering community, Edsger W. Dijkstra's dictum *Program testing can be used to show the presence of bugs, but never to show their absence!*[1] has been replaced by insights similar to Marie-Claude Gaudel's idea that *Testing can be formal too*[2].

Although testing has been studied in year one and two, key concepts are worth revising. Testing usually is a three phase activity:

Phase 1: Definition of the system under test (SUT) (e.g., a single Java method) and the quality to be established (e.g. functional correctness).

Here a concrete example – a Java method `unknown`

```
/**
* unknown multiplies two numbers
* @param a >= 0, a natural number
* @param b >= 0, a natural number
* @return the natural number a*b
*/
public static int unknown(int a, int b) {
```

[1] Notes On Structured Programming, 1972.
[2] So the title of her seminal paper in the Springer's LNCS 915 in 1995.

```
    int d = 0;
    int i=0;
    while (i < a) {
        d = d + b;
        i++;
    }
    if (a=0) {
        return -1;
    } else {
        return d;
    }
}
```

which shall be functionally correct w.r.t. the computational problem
Multiplication:
Input: natural numbers a, b with $0 \leq a, b \leq 9$
Output: the natural number $a * b$

Phase 2: Writing of a test suite according to a test selection strategy – e.g., Boundary Value Testing.

In the example of the Java method above this is a table where each row represents a test case. Such a test case has a name, collects the inputs, and also says which result is expected. Boundary Value Testing prescribes which input values to select. The output values are then produced manually according to the computational problem at hand:

Name	input 1	input 2	expected result
T1	0	5	0
T2	1	5	5
T3	4	5	20
T4	8	5	40
T5	9	5	45
T6	0	4	0
T7	1	4	4
T8	8	4	32
T9	9	4	36

Phase 3: The test suite is executed on the SUT. In a first step the actual responses of the SUT are documented. In a second step the actual result is compared with the expected result. This leads to the test verdict in {pass, fail}.

Name	input 1	input 2	expected result	actual result	verdict
T1	0	5	0	-1	fail
T2	1	5	5	5	pass
T3	4	5	20	20	pass
T4	8	5	40	40	pass
T5	9	5	45	45	pass
T6	0	4	0	-1	fail
T7	1	4	4	4	pass
T8	8	4	32	32	pass
T9	9	4	36	36	pass

The seemingly simple question if a test has passed or failed is famous as the test oracle problem which in general is undecidable.

For a complex system, testing usually takes places on different levels: e.g., on system, module, and unit level. The qualities to be established range from usability over performance to correct implementation of interfaces to functional correctness.

The special problems of testing Object Oriented software w.r.t. the levels of testing are discussed, e.g., in P C Jorgensen: *Software Testing: A Craftman's Approach.* 3^{rd} Edition, CRC Press, 2008.

Test selection methods are generally divided in Black-box testing and White-box testing. The test suite above demonstrated the Black-box approach in terms of Boundary Value Testing; below there is an example of tool support for White-box testing.

Although beyond the scope of most projects, for critical systems, the authorities often prescribe testing according to a specific test selection method. For example, the Modified Condition/Decision Coverage (MC/DC) is prescribed for Avionics, e.g., in DO-178C:

- DO-178C, Software Considerations in Airborne Systems and Equipment Certification is a guidance for software development published by RTCA, Incorporated. The standard was developed by RTCA and EUROCAE. The FAA accepts use of DO-178C as a means of certifying software in avionics.

- The Federal Aviation Administration (FAA) is an agency of the U.S. Department of Transportation with authority to regulate and oversee all aspects of civil aviation in the U.S.

Usually, test documentation is a challenge: how can a company convince the authorities that

- the tests actually have been performed, and

- with the actual results as stated?

It is easy to forge test results. The table above, in phase 3, is pure fiction: the author of this document sits at the moment at London Paddington train station with 5 more hrs of travel left to get to Swansea, and – thanks to a nice pint of London Pride – is far too lazy to actually test the Java method, and thus forges the table shown above.

As test results are so easy to forge, students are in a troublesome situation: how can they provide evidence that testing has happened? Test scripts and run time protocols in the appendix are one possibility, below there is an idea how screen-shots might help.

Tool support There are various tools supporting testing. In the case of Java, e.g., "EclEmma" (see e.g. `http://update.eclemma.org/`) supports White-box testing for test suites formulated in JUnit. Figure 7 presents a screen-shot of this tool: the sub-window on the upper left shows which tests (e.g., `clip1`) have been executed; the "tick" to the left of `clip1` indicates that this test was successful. The sub-window with the code shows which parts of the code have been covered by the test suite: green[3] indicates that this line has been executed, red[4] says that none of the tests has lead to the execution of this line. The sub-window on the lower right, finally, displays the coverage in percentage for the two classes `Formulas.java` and `FormulasTest.java`.

Thus, EclEmma not only supports the analysis if a test suite covers a given program according to some criterion; EclEmma can also be used for *test documentation*: a careful selection of screen-shots can serve as evidence that

- testing actually happened, and
- the system passed certain tests.

Evidence of testing in the various documents In the Initial Document the students will schedule time for all three phases of testing. Possibly, the students will develop first ideas, which levels of testing will be useful, what the test aims are, and what test selection methods will be appropriate. The Dissertation will reflect on the initial set-up, present the results testing, and – in the appendix – provide evidence that testing actually happened.

9 Writing documents

The department is relatively lax w.r.t. the question in which format documents are written: they shall be readable and nice looking. The famous slogan "content counts" applies here as well. Consequently, you are free to use your most beloved desktop publishing system to produce your documents: be it TEX, LATEX, VTEX, Word, whatever you want. We actually don't care, but note that we will enforce a pdf output for our electronic submissions (free pdf printers can be downloaded on Windows platforms if needed). If you really press us hard: in technical terms readability boils down to 11pt fonts and single spaced formatting. Double sided printing of documents is welcome.

What we actually care about, however, is that your writing is up to scientific standard. Essentially this requires the mastering of three "disciplines":

1. Writing clear, easy to understand, and correct English.
2. Putting things into context.
3. Organising your document in an adequate way.

In the following, we will discuss these points to some detail – but first the hint:

[3] In this black and white print-out: light grey.
[4] In this black and white print-out: dark grey.

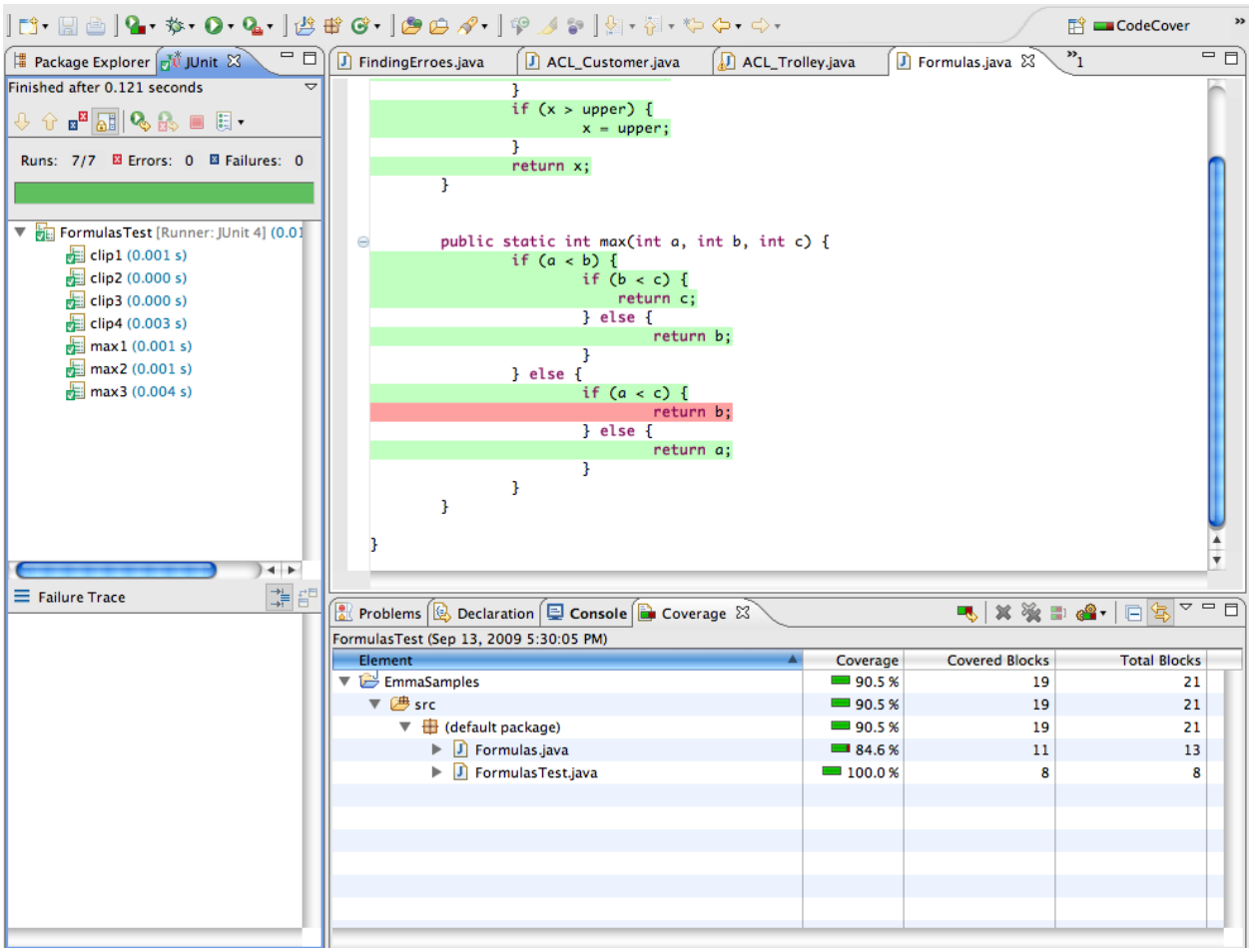


Figure 7: A screen-shot of running EclEmma.

Have test readers! Writing is a hard, time consuming, and error prone task. You will need feedback from test readers (friends, fellow students, lecturing staff, ...) in order to obtain a decent document. If your offer to test read one (or some) of your fellow student's documents, they will (or should) do the same for you.

Your reaction on feedback should be: yes, thank you for the hint – not: yes, but what I meant was ... Simply take the point: If things had been clear, your test readers would not have asked for clarification.

9.1 Writing good English

First, there should be a little bit of a disclaimer here: This is something that should have been learned in previous studies (e.g., school) and staff members shall only provide limited support there. It is however your job to try to do your best. If you are a native speaker, it does not give a good impression at all about the effort made. If you are not, unconsciously, people may be more lenient. Still, there is always a point where too many mistakes will give the reader a bad taste.

Spell checkers The easiest rule on proper writing is:

Your document should not include any spelling mistakes!

Please use a spell checker. A document with two or three spelling errors per pages is unacceptable and will obtain a reduced mark.

However, be aware that spell checking does not solve all your problems because it does not check the context in which you used a word – so it is still necessary to carefully proof read.

Grammar and complete sentences Astonishingly, many students think that fragments like "Pint of beer. Sunshine. Life good." are well written English. Grammar tells you: these sentences lack a verb. Please check that your sentences are complete and grammatically correct.

Clear and crisp sentences Are you in doubt if your formulations are crisp? Simply count the number of words between two full stops. A rule of thumb say: Any sentence over, say, 15 words fails to come to the point. Feel free to apply this rule to this paragraph :-)

Another hint in this direction is the following rule:

One thought - one sentence.

Here a concrete example from a – still unsubmitted – thesis:

> *JavaCC 5.0 is available for free download under the Berkeley Software Distribution License and as a parser generator, it takes as its input a grammar file with a .jj extension and it can be run from a command line with the javacc command.*

Here we have three thoughts in one sentence: (1) Is JavaCC free to use? (2) What input does JavaCC expect? (3) How to call JavaCC? Forming three sentences improves the text considerably:

> *JavaCC 5.0 is available for free download under the Berkeley Software Distribution License. As a parser generator, JavaCC takes as its input a grammar file with a .jj extension. JavaCC can be run from a command line with the javacc command.*

This still falls short from being perfect, however, some clarity has been gained.

Figures and text Any figure needs an explanation. It is good style to explain it textually in a step by step manner. For an example, see the explanation of Figure 7 in Section 8.6.

Sentences end with a full stop :-) Don't ask me why, however, a fact is: Especially in itemised lists, students tend to omit the full stop at teh end of sentences.

Connecting texts It is good style to have a text connecting different structuring elements. Here a bad example from real life:

> *2.1 Functional Requirements*
> *- The application should be able to calculate ...*
> *- It should be able to indicate ...*
> *- It should be able to visualise ...*

In this example, two text structuring elements stand unconnected against each other: The first element is the headline "2.1 Functional Requirements", the second element is an itemised list. A simple sentence can repair this and improve the flow of reading:

> *2.1 Functional Requirements*
> *The final product of our project shall have the following properties:*
> *- The application should be able to calculate ...*
> *- It should be able to indicate ...*
> *- It should be able to visualise ...*

Variation in the formulations The above example with its triple "should be able to" naturally leads to the rule:

Vary your formulations!

It simply is boring to read the same phrase over and over again. In the above example a simple "factoring out" could have helped:

> *2.1 Functional Requirements*
> *The final product of our project shall be able to*
> *- calculate ...*
> *- indicate ...*
> *- visualise ...*

Spacing There are standard rules for white spaces around punctuation marks. Here we compile a few:

Comma, colon, full stop, semicolon No white space before the punctuation mark, one white space after the punctuation mark.

Opening brackets One white space before an opening bracket, no white space after an opening bracket.

Closing brackets No white space before a closing bracket, one white space after a closing bracket.

Dash One white space before a dash, one white space after a dash.

Capitalisation The words figure, table, listing etc. start with a lower-case letter. However, the moment one refers to a specific figure, say we write "Figure 1 shows the graphical interface of our tool.", we speak about a specific figure – thus, in this context the word figure has to start with an upper-case letter.

Writing on the margin It is tempting, isn't it? The formula, the word, the figure, whatever it is just does not want to fit into the page format. Ahhhh, and there is this nice empty space on the margin.

The simple rule is: **Don't!** It's unprofessional, and gives the impression of sloppy work.

9.2 Addressing the scientific context

Whatever you project is about, there will have been forerunners on your topic. Most probably other researchers are working on your topic as well. You are not alone on this world ... As a scientist, you have to:

- acknowledge these contributions.
- put the specifics of your approach into this context.

To this end it is necessary to refer to these publications. This will allow readers to look up these contributions and to work out, if they agree with the way you tell the story.

Your work does not start from scratch: Your project will use libraries, tools, theoretical frameworks, etc to quite an extent. In your documents, you will have to discuss them and give proper references to them. Again, the purpose of the references is to allow readers to look up these contributions and to work out, if they agree with the way you tell the story.

This brings us to the boring, however, important question of how to write references. As references shall allow readers to find a document, you will have to collect the right bibliographical data. Although you will have written documents containing references before, this topic is important enough for revision. Here you can use Appendix A as a guideline which bibliographical data is needed.

Writing References The first insight is: Documents come in different flavours. Thus, they require different collections of bibliographical data. Appendix A discusses this in the context of the program bibtex. Without implying that you should use this specific program, this discussion will guide you to select the right data for your references. Please note though that, although we don't require you use it, bibtex (and tools like it) will help you manage references, reduce the workload in dealing with them properly, and reduce the chance of errors.

Section B.2.1 lists different "entry types": a piece of bibliographical data can be necessary to find a document – then it is required; a piece of bibliographical data can be useful to find a document – then it is optional; a piece of bibliographical data can be pointless to find a document – then you better don't include it.

Following this, the section classifies documents into the type "article", "book", "booklet", etc. For each type of document, a different set of data is required, while other data is optional or ignored.

Finally, in Section B.2.2 you find explanations of the bibliographical data itself.

Following this approach, you should have no trouble to compile proper references. As an example we demonstrate how to compile the reference for the book from which we scanned in these instructions.

The document is a *book*. Consequently, it is required to write down *author, title, publisher, and year*. For using the LaTeX bibliography database, we could compile this information into a *bibtex entry*:

```
@Book{lamport94,
  author =       {Lamport, Leslie},
  title =        {LaTex},
  publisher =    {Addison-Wesley},
  year =         {1994},
}
```

In the printed version, one would compile this as

[Lam94] L Lamport: *LaTeX*. Addison-Wesley, 1994.

If you are not using bibtex (or similar) you can just enter this final form directly (though you may well end up entering it more than once in the different documents you write).

References to the Internet Be careful with references to the Internet. They might be cumbersome for two reasons:

1. One issue is the question of the quality of these web-pages. Often, web-pages get things badly wrong, see Figure 8. Books, journals, and conference proceedings undergo a process of review which increases their credibility. Such a process is not present in the case of web-pages. Thus, you better do a "plausibility check" if one actually can believe the content of a web-page. For example, finding some other source that confirms it. (And, of course, if that other source is a book, journal or conference proceedings, you might as well use it *instead* of the web page.)

2. Web-pages are bound to change. Thus it is not clear if readers of your document will actually be able to follow up a reference you are giving to a web-page: The page content might have changed, the page might even not exist any longer. Thus, the basic rule for web-references is:

   Whenever you refer to the Internet
   add the date at which you accessed the web-page.

9.3 Organisation of your documents

Things are simple. A proper document consists of (in this order)

1. a title page.

2. a table of contents.

Figure 8: Where the Internet got it wrong ...

3. an introduction.
4. several sections for the main body of your work.
5. a summary or conclusion.
6. references.
7. possibly an appendix.

In the following, we will go over some of these elements.

Title-page For the Initial Document you are free to design your own title page. The author of this document is a minimalist when it comes to title pages. Take a look at the title page of this document to get an example of what he likes.

Here is what you need to put on a title page: Title of the document, name of the author, date when it was written. As subtitle you should include if it is the Initial Document. For marking purposes, the student number is also helpful and necessary (e.g., document delivered to the wrong person).

For the dissertation, the department has a prescribed title page - see Appendix B.1.

For all three documents, it is important that you find a title which is

- an eye catcher.
- however still is adequate.

Coming up with a good title for a project is a non-trivial task. Thus, you better schedule some time for it and start thinking right now.

Table of contents The Table of contents is easy enough to produce: you just list all chapter names and add on which page these can be found (and tools like LaTeX can do it automatically for you). You have an example of this at the beginning of this document. The pitfall to avoid is: Do not list the Table of contents in the Table of contents.

Although easy to produce, the Table of contents is far more than just an index: It allows the reader to grasp the structure of your document. Thus, you better have "speaking", meaningful chapter titles. Furthermore, the Table of contents clearly tells how you balance things: What is the proportion between background and own work? Which topics do you write lots about? Which topics do you neglect? In this sense you can control how well your document mirrors your intentions.

Introduction Writing a good introduction is an art! Your supervisor and second marker will read your document thoroughly. However, other people may only read Title, Introduction, and Summary/Conclusion of your document (and the same is likely to be true of other documents you write in your future career). Thus, you better shine in them.

Introductions serve various purposes: They shall position your work within Computer Science; they shall make the reader interested in your document; they shall inform the reader about the contents of your document.

Let's have a look at an example for this. The Figures 9 and 10 show the first two pages of a conference article. The first paragraph positions the paper and also tries to get the reader interested. Paragraphs two and three inform on the background of this work. Paragraph two addresses the question of the specification languages involved. Paragraph three discusses the testing approach on which this paper is based. Paragraph four finally gives an idea of what the paper will be about. The final paragraph is a spelt-out Table of Contents.

The rules of the game are: In your documents you are bound to have a first paragraph that positions your document; you are also bound to have a last paragraph which spells out the table of contents and briefly states what is in each chapter (project documents are not mystery novels – there is no need to keep what happens next a surprise). The other paragraphs are "free" – where you have to work out how to get the reader interested and how to inform the reader what your document is about.

Appendix The appendix comprises of information which is useful but not essential for understanding of your document. Every Program Listing or Figure or List of commands or Table or ... which is longer than, say, one page belongs into the appendix. "Huge" elements destroy the flow of reading – as you probably have experienced with Figures 9 and 10. If you want to demonstrate a crucial point in, say, a code example, this will be possible in about 10 lines of code: you just have to identify the essential lines and explain their context. For the interested reader, you can provide the full code example in an appendix.

10 Making presentations

Not many things are more debatable than what makes a good presentation. Ask three people and you will get four different opinions. However, there is also a profession devoted to the design of presentations. Some of the design rules that this profession developed shall be discussed here.

Nowadays, the standard format of a presentation in Computer Science is a talk supported by slides shown from a computer. Other formats would be a talk involving a blackboard, or even a

Towards formal testing of jet engine Rolls-Royce BR725*

Greg Holland[1], Temesghen Kahsai[2],
Markus Roggenbach[2], Bernd-Holger Schlingloff[3]

[1] Rolls-Royce Plc. Derby, UK.
[2] Department of Computer Science, Swansea University, UK.
[3] Humboldt University Berlin, Fraunhofer FIRST, Germany.

Abstract. The Rolls-Royce BR725 is a newly designed jet engine for ultra-long-range and high-speed business jets. In this paper we apply our theory of formal testing [5,6] to the starting system of the Rolls-Royce BR725 control software. To this end we model the system in CSP, evaluate test suites against the formal model, and finally execute test suites in an in-the-loop setting of the SUT. This case study demonstrates the applicability of our testing approach to industrial systems: it scales up to real world applications and it potentially fits into current verification and quality assurance processes, as e.g., in place at Rolls-Royce.

1 Introduction

Jet engines belong to the safety critical systems of an air plane. Their control software can be classified as a reactive system: it accepts commands from the pilot, receives status messages from the airframe and the engine sensors, and issues commands to the engine. Here, we report on the successful application of our theory of specification based testing [5,6] to such systems.

Our testing theory has been developed for the formal specification language CSP-CASL [10]. CSP-CASL allows to formalize systems in a combined algebraic / process algebraic notation. To this end it integrates the process algebra CSP [3,11] and the algebraic specification language CASL [8]. In the context of this paper we restrict CSP-CASL to (a sub-language of) CSP_M, the machine-readable version of CSP.

Figure 1 shows our testing approach in a nutshell. *Specification*, *Implementation* and *Test Cases* are mutually related artifacts. *Specifications* and *Test Cases* are written in CSP-CASL, the *Implementation* is treated as a black box. *Test cases* can be constructed either from the specification – as shown in the triangle – or independently from it. The specification determines the alphabet of the test suite, and the expected result of each test case. The expected result is coded in a colouring scheme of test cases. If a test case is constructed which checks for the presence of a required feature (according to the specification), we define its colour to be *green*. If a test case checks for the absence of some unwanted behaviour, we say that it has the colour *red*. If the specification does neither require nor disallow the behaviour tested by the test case, i.e., if a System Under Test (SUT) may or may not implement this behaviour, the colour of the test case is

* We would like to thank Rolls-Royce for supporting the internship of the second author. This work was supported by EPSRC under the grant EP/D037212/1.

Figure 9: An Introduction - first page.

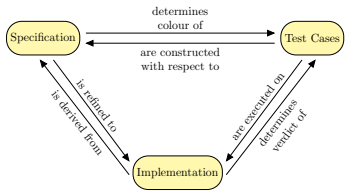


Fig. 1. Validation triangle

defined to be *yellow*. During the execution of a test on a particular SUT, the *verdict* is determined by comparing the colour of the test case with the actual behaviour. A test *fails*, if the colour of the test case is green but the SUT does not exhibit this behaviour, or if the colour is red but the behaviour can be observed in the SUT. The execution of a yellow test case yields an inconclusive verdict. Otherwise, the test passes.

Here, we apply our theory to the starting system of Rolls-Royce BR725 [4] software control. The BR725 is a newly designed jet engine for ultra-long-range and high-speed business jets. It is part of the BR700 family. We model the starting system in CSP and validate our model using the CSP simulator PROBE. We then evaluate the test suites against the formal model. Such evaluation is done using the model checker FDR2. Part of the test suites is inspired by existing test cases of the BR700 family jet engines. We execute our test suite in an in-the-loop setting on the so-called "rig". This puts the engine control system through test scenarios identical to those carried out in engine test stand testing, however with considerable lower cost, reduced risk, and less burden on human and mechanical resources.

Outline In Section 2 we give an overview of the test evaluation theory from [5] and a brief introduction to CSP_M and FDR2. In Section 3 we describe a control system of a jet engine in general and the starting system of the Rolls-Royce BR725 jet engine in particular. We also show how we model the latter in CSP_M. Subsequently, in Section 4 we show how we evaluate and execute test cases.

2 Testing in CSP-CASL

In this section we give an overview of the theory of specification-based testing presented in [5,6]. The theory is based on the specification language CSP-CASL [10] which allows to formalize computational system in a combined algebraic / process algebraic notation. CSP-CASL uses the process algebra CSP [3,11] for the modeling of reactive behaviour, whereas the communicated data is specified in CASL [8]. CSP-CASL has been deployed in the modeling [2] and verification [4] of an electronic payment system.

In [5], a theory for the evaluation of test cases with respect to CSP-CASL specifications has been developed. In summary, the main benefits of this theory are as follows.

[4] In the rest of the paper we will refer to this engine type simply as BR725.

Figure 10: An Introduction – second page.

talk without any medium supporting it. For the Gregynog presentation students are free to choose a format of their own, however, students are strongly advised to stick to the standard format: it simply is the easiest of all.

Although you have already made presentations, and been taught how to do it, this, like referencing, is important enough for revision.

10.1 How to make slides look good

There are many rules of good design for slides. Here, we discuss a basic selection of topics.

Font size: The foremost design rule for slides is:

Use a large enough font!

It does not make much sense to provide material on slides that in the end no one can read. Concerning readability, one should also think about colours: in projection, colours often look very different from the colours on your screen. As it usually is impossible to check with the projector first, the rule of thumb is: use colours which have a big contrast to the background.

Colour: Life is dull without colour, however colour needs a meaning in a talk. Thus, one should develop a clear colour scheme with a clear meaning. Here the sample scheme from our slides:

Colour scheme

headlines	–	*green*
emphasised words	–	*red*
footers	–	*blue*
normal text	–	*black*
sub items	–	*grey*

A clear and simple colour scheme improves the readability of slides. One can of course add a font selection scheme in the same way. This will allow then, e.g., to encode "type information":

Font selection scheme

standard text	–	in normal font
mathematical formulae	–	*in italics*
code examples	–	`in teletype font`
names	–	IN SMALL CAPS

These schemes should then consequently be applied throughout the talk.

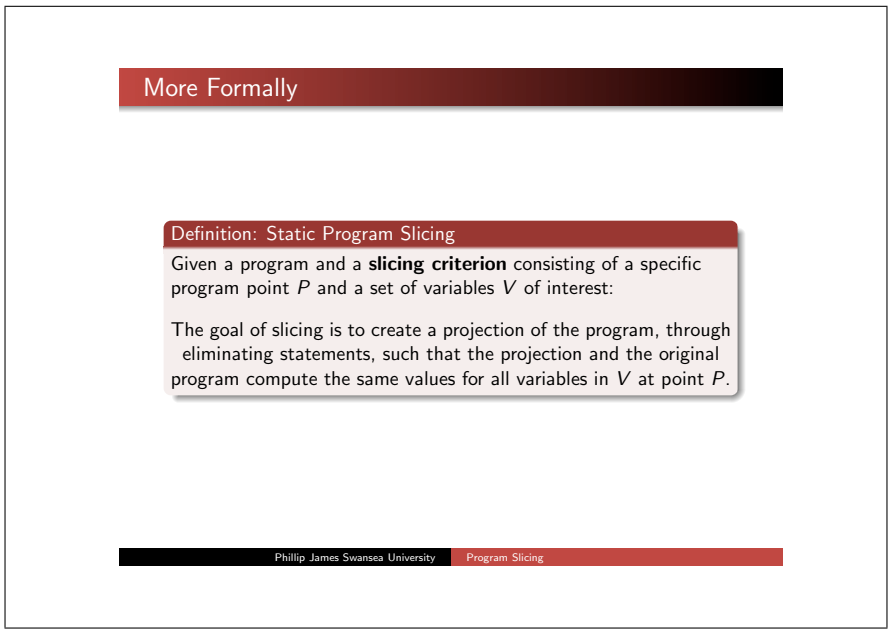
7 × 7 rule: Slides shall not divert the attention from the presenter. Thus, they better be easy to grasp. To this end, professionals in slide design came up with the following rule:

The 7 × 7 rule

On one slide:
No more than 7 lines of text,
no more than 7 words per line.

The rationale behind this rule is that people can usually comprehend up to seven different items at a glance. With a slide following the 7 × 7 rule this means: the slide structure is clear after two "glances". The first glance allows to comprehend the first line, the second glance gives the slide structure. After this, the attention of the audience is back again with the speaker.

Catch phrases: Novels can be entertaining, however, writing novels on slides does not work out. A slide should offer catch phrases only. Here a negative example, in "novel style":



Here the transformation to catch phrase style – without nice formatting:

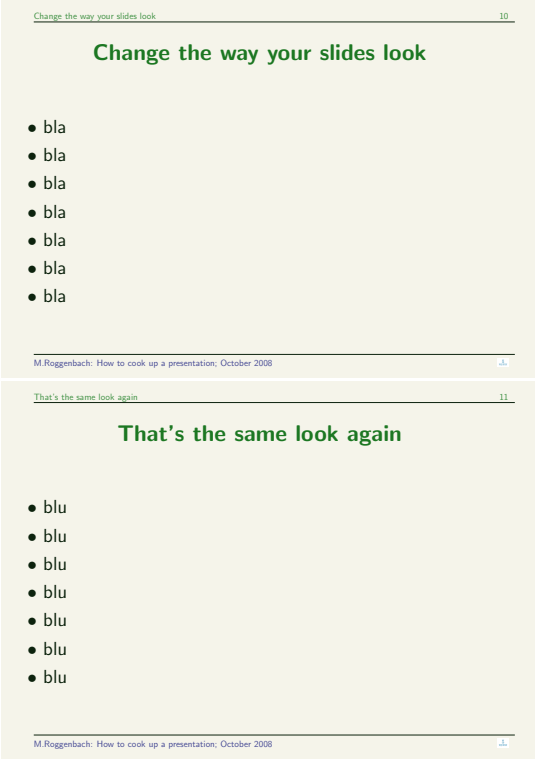
```
            Static Program Slicing

Given:
  Program P
  Slicing Criterium: (line number, set of variables)

Wanted:
  Program P' such that
  - P' = projection (P)
  - values(P', line number) = values (P, line number)
```

With the support of the above slide Phil could have told the same novel – however, this time orally and also with the attention of the audience. Anyway, it is worthwhile to note that Phil in the end got a mark of 90% – which is a nice proof that content counts!

Pictures: Pictures are simply great on slides. As the proverb says:

A Picture Says More Than A Thousand Words.

Change in the overall layout Having slides all the way through in an identical layout is simply boring. You want a proof? Here three samples (and believe me: since the invention of Powerpoint 80% of all talks look exactly this way!)



No wonder that in such talks half the audience falls asleep.

Spelling Yet another quality criterion is correct spelling. No more to say than:

At a tme
were we have spellchekers availabel
speling errors
relly
ar cumbersame.

Relating slides and speech It is a good idea to produce a talk-plan for every slide! This might look as follows:

Here is the slide	1-st item to say 2-nd item to say ... n-th item to say

Having such a plan you can now check:

1. Are there elements on the slide with nothing to say about?
2. Are there things you want to say without visual support?

Slides support the talk The presentation should be far more than what is written on the slides. In a good presentation the slide provides some keywords, however, the presenter tells the exciting story.

Keep it Simple Some people bring in large amounts of notes, or prompt cards, to remind them what to say. They usually don't refer to them at all, or if they do they remember half way through the talk and can't easily find the bit they need. It's better to simply to know the talk well, and make sure you embed "cues" on each slide to prompt you to talk about each aspect as your reach it.

10.2 How to organise a talk

Talks need a structure. The standard organisation looks as follows:

- Title slide
- Motivation (if appropriate)
- Table of contents
- 3 – 4 chapters
- Summary
- Future Work (if appropriate)

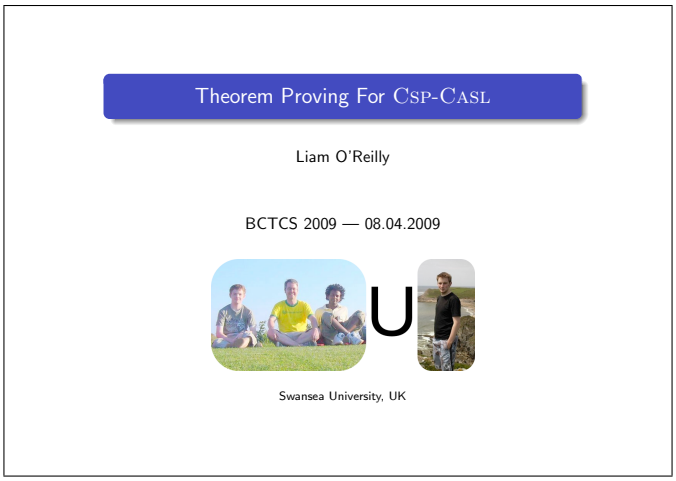
There is of course a reason why you should organise your talk in no more than three or four chapters: Everyone in the audience will be able to remember three or four items; any more demanding structuring will be lost to the audience.

In the following, we will discuss these topics in some detail.

The title slide What you should put on it is

- a catching title – in a big font
- your name – in a slightly smaller font
- the occasion – in normal font size
- what institution you come from – in normal font size – if this is not clear anyway.

And believe it or not: there is really, really, really no need for anything more! Ok, ok – as you insist: maybe a picture could help. But that's it. Here a sample title slide:



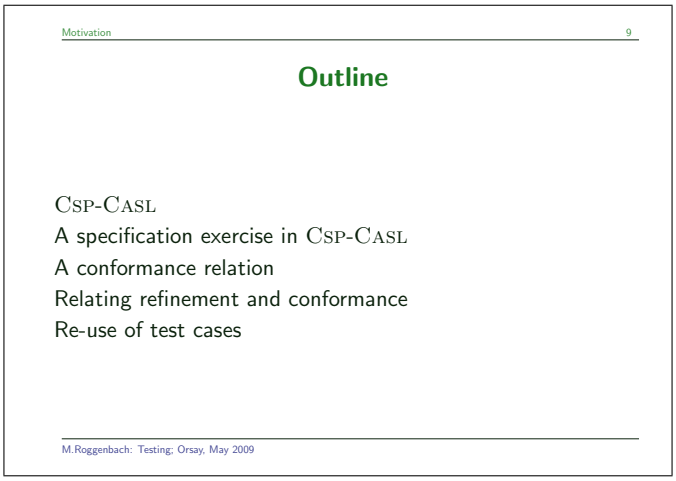
Motivation That's the place to state the initial problem – if this is helpful in order to explain the table of contents. Usually it is better not to give a motivation but to directly jump into the talk. Anyway, to give a nice motivation, a picture is always a good start ...

Table of Contents This is the place where you first give a structure of your thoughts to the audience. Think of your audience as active listeners: they want to learn about your subject. This will be easier, when you first give them some landmarks for the, in the beginning for the audience, vast field that you will fill with detail.

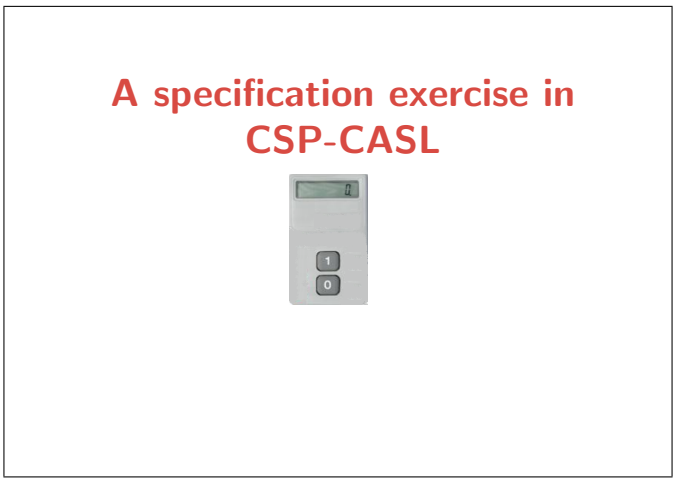
In the table of contents you simple list the titles of the 3 or 4 chapters that your talk has. Here you should have better titles than just "Introduction" or "Background". As these titles apply to all talks, you are giving no information: i.e., you are wasting your time as well as the time of the audience.

And please, pretty pretty please: Absolutely never, ever, under any circumstances whatsoever include *table of contents, summary, conclusion* or *future work* into your table of contents. It is the same story again: as every talk is supposed to have these items, it is a waste of time and therefore considered bad style to mention them. The often heard sentence "And finally I conclude my presentation with the conclusion." illustrates this idiocy nicely.

Here a sample of a table of contents:



Communicating the talk's structure During your talk it is important to communicate where you are within the initially announced structure. To this end you can have a separation slide between chapters, on which you simply present the next chapter's title, maybe combined with a picture:



Another approach would be to show the table of contents once more, with the next chapter emphasised.

Whatever technique you prefer, the change from one chapter to the next one is the time to apply the story teller's rule[5]:

- Say what you have done so far.
- Say what you are going to do.
- Do it.

Summary/Conclusion This part of the talk is your last chance to sell your work! You better make the best out of it. The central question is: what was your talk really about? You need to formulate the three or four points that you want to be remembered.

Future work Oh dear, that's a hard one. Its a questions of balance. If there is "too much future work", you essentially communicate that you did not do much. If there is "no future work", then you communicate: your topic has no future.

However, as your project presentation at Gregynog is about an ongoing project, this balance should be easy to find: you are simply in the beginning, with the major body of work still to be done.

10.3 The time issue

Usually, you are given a fixed time slot for your presentation, say, x minutes you have. But how can you make sure that you prepare a presentation of the length required?

[5]Here adapted to this concrete situation. Originally it states: say what you are going to do, do it, say what you have done.

Predicting the length of a presentation A rule of thumb says:

The presentation of one content slide takes about 2 to 3 minutes.

This rule allows you to easily compute the number of slides for your Gregynog presentation: you have 15 minutes, assume 2 minutes of presentation time per slide, and you end up with about 8 slides for the content. Then you add a title slide, a table of content slide, the separation slides, a slide for the summary, and a slide of the future work. There you are :-)

Staying in time during the presentation In order to stay in time, you will need your own watch. It is one of the last unresolved secrets of mankind why rooms for presentations never have a clock visible to the speaker, or, if they have, why this clock never works. Assuming you did bring your own timing device you will do well, provided you make a time plan beforehand. Such a time plan needs to be verified in a 'dress rehearsal'. This needs to be a real one in front of an audience. In a mental walk through you will always be faster. Back to the time plan and the watch: they allow you to control time during the talk. Prepared 'short-cuts', but also 'hidden' slides for additional material will allow you to adapt your talk to the concrete situation.

Everyone will be happy should your talk finish slightly under time. However, if you finish far too early the overall impression will be: you had no message to tell. Should you finish far too late, you will be considered a bad speaker.

10.4 Performance preparations

Rehearsals help How many presentations have you given yet? Chances are, it has not been many .... Thus, it is best to have a dress rehearsal in front of some friends and/or fellow third year students. As the proverb says:

Practice makes perfect!

Most probably you will learn from such a rehearsal:

- Your presentation is too long.
- Supposedly clear topics are simply not understood by the audience.

This means: time for a re-design of your talk.

First sentences Nearly everyone gets nervous when presenting in front of an audience. Usually, the peak of nervousness is just at the beginning of the talk – exactly when the audience collects the famous "first impressions". Thus, the following is good practice:

Learn the first few sentences by heart.
This avoids stupid mistakes due to nerves.

10.5 And finally: The performance

Although you probably are uncomfortable with giving a presentation (most students are) – at least on an intellectual level you should realise that you have all reason to be grateful to your audience. In the end everyone in the audience takes the time to listen to your presentation, they could be doing so many other things instead. Thus, the first rule is:

Be kind to your audience!

Concretely, this means:

- Say "hello" in the beginning.
- Answer kindly any questions arising.
- Thank the audience for its attention at the end.

On stage You might wonder about the choice of words: "performance", "stage" – but this is really what a presentation is about. To a certain extend you are an actor when giving a presentation. To give you an extreme example:

Walter H. G. Lewin, 71, a physics professor, has long had a cult following at M.I.T. And he has now emerged as an international Internet guru, thanks to the global classroom the institute created to spread knowledge through cyberspace.

Professor Lewins videotaped physics lectures, free online on the OpenCourseWare of the Massachusetts Institute of Technology, have won him devotees across the country and beyond who stuff his e-mail in-box with praise.

In his lectures at ocw.mit.edu, Professor Lewin beats a student with cat fur to demonstrate electrostatics. Wearing shorts, sandals with socks and a pith helmet nerd safari garb he fires a cannon loaded with a golf ball at a stuffed monkey wearing a bulletproof vest to demonstrate the trajectories of objects in free fall.

He rides a fire-extinguisher-propelled tricycle across his classroom to show how a rocket lifts off. [New York Times]

Prof Lewin is known to stage his lectures to the tiniest detail. That's what makes him so good. Well, no one expects such a performance from you. However, you should be aware of the following points:

Voice volume You should be clearly audible for everyone in the room. There are people who speak quietly as they are a bit shy. Others, thanks to the excitement of the moment, start to shout. You should try to find out what type you are and adapt accordingly.

Body language You sell your presentation also with the way your body speaks: are you confident about this part of your talk, do you have doubts, are you open to this question, or does this question really get on your nerves?

The basic advice for beginners is: find a stable position in which you feel comfortable and come regularly back to this position during your talk.

Some people are "walkers": they never stand still, run from one corner of the room to the other, back and forth, never find a position. This is often perceived as lack of confidence.

Other people are more static: they stand in one position all the time. While this often gives the impression of a confident person, the no-movement can be seen as inflexibility and overall makes the presentation boring.

Eye contact The audience wants to be addressed. Thus, you have to "look after" them. Concretely, everyone should get the feeling that you look at him/her – at least from time to time. One technique to achieve this is to look over the audience in "diagonals": starting in the front left moving to the back right, and then starting from the front right moving to the back left. Another technique is to have "fixed people" distributed in the audience: you look at them from time to time.

This eye contact provides you also with important feedback on your talk: Are you too fast, or too demanding? Is the audience still with you? Is it time to ask a question?

Stage-preparations With the previous remarks, it should not come as a surprise that in a presentation your "stage" is nothing but a working place that better functions in an optimal way while you are performing. Thus, before the presentation you should check the set-up: Where will you stand? How will you change the slides? How can you point to a slide? Where can you stand without obscuring the slides?

It is also vital to remove potential obstacles (cables, chairs, tables, ...) in time. You have no idea what funny things you can find on your stage!

Last but not least: Work out beforehand that your slides/programmes/tools actually run on the machine you will be using. Be aware that in the end the audience will hold you responsible if things don't work out – independent of whose mistake it was in the end. You are the host, they are here for your sake, so you better make things right for them.

10.6 Content counts!

In the previous sections we discussed ways of how to prepare a good presentation. It should be noted that they concern the quality of the *form* of the presentation. The best form, however, does not help if there is no content that you want to tell. Thus, here the last, and maybe, most important rule:

Content counts!

11 Designing Posters

Before designing a poster, it is important to understand the communication situation at the Project Demonstration Fair, see Figure 11 for typical room layout.

Figure 4 gives you an additional idea how things probably will look like.

You will be busy with the Project Demonstration Fair for a whole afternoon. A typical time-plan for the fair is:

14.00 – 15.30 Setting up (in assigned time slots)

16.00 – 18.00 Fair Part I (with Tea/Coffee)
Audience: 1st year students; 2nd year students; lecturing staff.

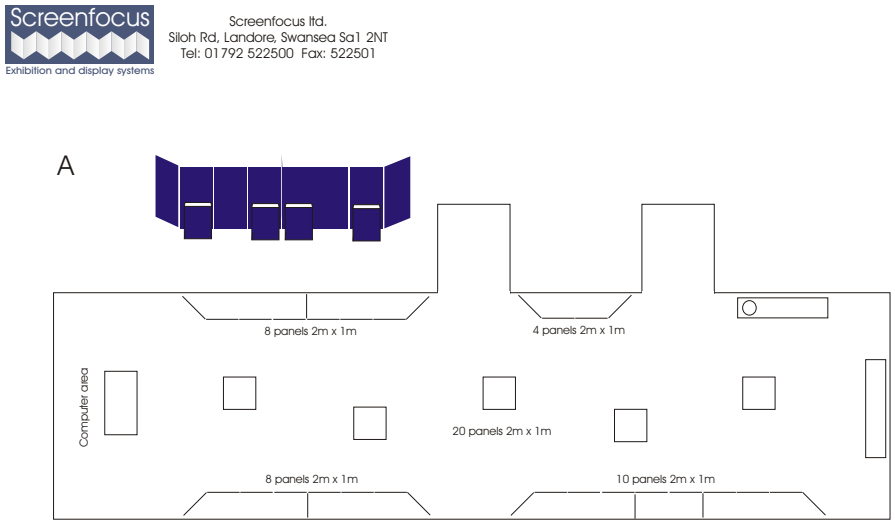


Figure 11: Room layout of the Project Fair.

18.00 – 19.30 Fair Part II (with dinner buffet)
Audience: MEng students; MSc students; Postgrads;
local IT industry; lecturing staff.

Note: after setting up your stand, you can take a break and return to the Project Demonstration Fair when it is opened. You will be required to be at the Fair for both parts.

11.1 Poster Design

Your poster will need to support the following communication situations:

Attract attention Even from distance your stand shall look unique and attract attention. This requires you to find a visible selling point. In 2009, Jon (below) did quite a good job in this respect:

Short meeting: 2 mins And there he is: your first customer. How will you explain your project to him? Well, hopefully you have listed the main points of your project somewhere. Chris (below) was quite good when it came to attracting attention. However, his poster did not offer much besides the eye catcher: Thus he had to explain all the details without support from his poster.

Even if the first attraction was there, not all customers will want to stay for longer. For these situations you need to have prepared a, say 2 minutes presentation that covers all major aspects of your project.

Long meeting: 6 mins Far more rewarding is if your customer spends a longer time for your stand. Then you need to have loads of interesting details at hand. You will want to be able to speak about all the tricky points, where you came up with excellent solutions.

With the limited space available it is a challenge to serve all these three requirements. In the following we will discuss these points in more detail.

What is your eye-catcher? You will need to find one (!!!) good picture that motivates or is related to or illustrates your project! Usually it takes long time to come up with an idea. Maybe you want to already start to think about this now?

Questions your poster should answer The poster is a summary of your project. To this end it needs to address the following points:

- What is the project about?
  (topic, motivation, similar projects, ...)
- What methods were used?
  (programming language, software tools, algorithms, theories, ...)
- How was the problem solved?
  (software architecture, own algorithms, code examples, ...)
- What are the results?
  (is the world a better place after this project?)

You have to prepare two sets of answers for each of them: one set for short meetings; another set for long meetings. Thus, it might be a good idea to have headlines in a big font – the version for the short meetings can have the the details in a smaller font, however still readable from 1 – 2 meters away.

Layout considerations You have space, however, not much! It will be $1m \times 1m$. Not more, not less. Jon, as you can see above, filled it with one big poster. You can have several small posters as Khyle:

Which 'tiling' do you like better? Either can be made to work well.

How to organise things on the poster(s)? It is tempting to prepare a poster like a talk, where the poster in the end looks like a sequence of slides. Although this is of course a legitimate solution, it usually makes a bad impression: the poster designer obviously confused the medium, or, even worse, simply turned a talk into a poster, i.e., she did not take the occasion seriously.

Usually, it is better to have the 'poster as one unit' which develops one big picture of your project. One possibility is to organise things in star form: the final product is in the middle and is surrounded by "aspects".

Mind Mapping The burning question is: How do develop such big picture? Here, the Mind Mapping method has proved to be an excellent technique. A good and short explanation by Tony Buzan can be found on youtube: `http://www.youtube.com/watch?v=MlabrWv25qQ`.

Layout: The usual design rules apply! Here just the most important ones:

- Large font.
- Clear colour scheme / font selection scheme.
- Catch phrases – not novels.
- Easy to understand layout.
- Have pictures.
- No deviating information ('grandmother's shoe size').

For the detailed discussion of all these points see Chapter 10 "Making presentations".

11.2 Computer demonstrations

The computer demonstration is obligatory at the Project Demonstration Fair:

Software projects
need to demonstrate a
'running system'.

Such a demonstration needs a good preparation. Before starting the software, you should tell your visitor:

- What you are going to demonstrate.
- Why it is valuable.
- What the effect will be.

and possibly asks whether he is familiar with some of the concepts exposed. Prepared in this way, your visitor will be able to understand what is going on.

11.3 Things to pep up your stand

Up to now, we have only discussed how you can come up with a poster. The next question is obviously what else can you do in order to make your stand and yourself more memorable. Here some ideas:

- Business cards,

- CV,
- Flyers about your project,
- Mascot (Erwin is already taken!),
- Sweets,
- ...

11.4 Posters – How to produce it?

After all these "theoretical" considerations let's get down to business. How can the actual design process take place?

Software (some ideas): There are many software packages available that allow one to design posters. Here some packages that have been used successfully in the past:

- inkscape (linux/windows/mac)
- Powerpoint
- word (word2pdf converter is installed on the departmental computers)
- photoshop (in case you own it – there are free alternatives)
- Illustrator (again there are free alternatives – inkscape above is one)

Options for printing Within the departmental, there are b/w printers in A4.

At your own costs, you can print your poster at professional print shops or using services offered via the internet.

12 Conclusion with doubts

So, that's it for the handbook. Hope it helps.

"What a waste of time!" was Roger H.'s reaction when he heard that Markus R. got appointed to write this handbook. And he continued: "Why would anyone expect a student to read such a handbook, if the student was not able to pick up this information in the lectures?" However, when he read it, he changed his mind:

```
Dear Markus,

Thanks for the copy of your $3^{rd}$ Year Project Handbook. After looking
at it, I must admit I was wrong to disparage the purpose of such a
book; you have put a lot of really useful information and advice in
it.

Yours,
 Roger
```

Acknowledgement

Many people commented on the handbook. My special thanks go to Markus Roggenbach who wrote it almost in its entirety, Neal Harman, Phil James, and Liam O'Reilly for going over it in a systematic way, and to Erwin R. Castebeiana (Jr) for all his acute remarks.

A Excerpts from L. Lamport: Latex. Addison Wesley, 1994.

Below is a description of how to use bibtex, which can greatly simplify managing references if you are using LaTeX.

B.2.1 Entry Types

When entering a reference in the database, the first thing to decide is what type of entry it is. No fixed classification scheme can be complete, but BibTeX provides enough entry types to handle almost any reference reasonably well.

References to different types of publications contain different information; a reference to a journal article might include the volume and number of the journal, which is usually not meaningful for a book. Therefore, database entries of different types have different fields. For each entry type, the fields are divided into three classes:

required Omitting the field will produce an error message and will occasionally result in a badly formatted bibliography entry. If the required information is not meaningful, you are using the wrong entry type. If the required information is meaningful but not needed—for example, because it is included in some other field—simply ignore the warning that BibTeX generates.

optional The field's information will be used if present, but can be omitted without causing any formatting problems. A reference should contain any information that might help the reader, so you should include the optional

field if it is applicable. (A nonstandard bibliography style might ignore an optional field when creating the reference-list entry.)

ignored The field is ignored. BibTeX ignores a field that is not required or optional, so you can include any fields you want in a `bib` file entry. It's a good idea to put all relevant information about a reference in its `bib` file entry—even information that may never appear in the bibliography. For example, if you want to keep an abstract of a paper in a computer file, put it in an `abstract` field in the paper's `bib` file entry. The `bib` file is likely to be as good a place as any for the abstract, and it is possible to design a bibliography style for printing selected abstracts.

Misspelling its name will cause a field to be ignored, so check the database entry if relevant information that you think is there does not appear in the reference-list entry.

The following are all the entry types, along with their required and optional fields, that are used by the standard bibliography styles. The meanings of the individual fields are explained in the next section. A particular bibliography style may ignore some optional fields in creating the reference. Remember that, when used in the `bib` file, the entry-type name is preceded by an `@` character.

article An article from a journal or magazine. Required fields: `author`, `title`, `journal`, `year`. Optional fields: `volume`, `number`, `pages`, `month`, `note`.

book A book with an explicit publisher. Required fields: `author` or `editor`, `title`, `publisher`, `year`. Optional fields: `volume` or `number`, `series`, `address`, `edition`, `month`, `note`.

booklet A work that is printed and bound, but without a named publisher or sponsoring institution. Required field: `title`. Optional fields: `author`, `howpublished`, `address`, `month`, `year`, `note`.

conference The same as **inproceedings**, included for compatibility with older versions.

inbook A part of a book, usually untitled; it may be a chapter (or other sectional unit) and/or a range of pages. Required fields: `author` or `editor`, `title`, `chapter` and/or `pages`, `publisher`, `year`. Optional fields: `volume` or `number`, `series`, `type`, `address`, `edition`, `month`, `note`.

incollection A part of a book with its own title. Required fields: `author`, `title`, `booktitle`, `publisher`, `year`. Optional fields: `editor`, `volume` or `number`, `series`, `type`, `chapter`, `pages`, `address`, `edition`, `month`, `note`.

inproceedings An article in a conference proceedings. Required fields: `author`, `title`, `booktitle`, `year`. Optional fields: `editor`, `volume` or `number`, `series`, `pages`, `address`, `month`, `organization`, `publisher`, `note`.

manual Technical documentation. Required field: `title`. Optional fields: `author`, `organization`, `address`, `edition`, `month`, `year`, `note`.

mastersthesis A master's thesis. Required fields: `author`, `title`, `school`, `year`. Optional fields: `type`, `address`, `month`, `note`.

misc Use this type when nothing else fits. Required fields: none. Optional fields: `author`, `title`, `howpublished`, `month`, `year`, `note`.

phdthesis A Ph.D. thesis. Required fields: `author`, `title`, `school`, `year`. Optional fields: `type`, `address`, `month`, `note`.

proceedings The proceedings of a conference. Required fields: `title`, `year`. Optional fields: `editor`, `volume` or `number`, `series`, `address`, `month`, `organization`, `publisher`, `note`.

techreport A report published by a school or other institution, usually numbered within a series. Required fields: `author`, `title`, `institution`, `year`. Optional fields: `type`, `number`, `address`, `month`, `note`.

unpublished A document with an author and title, but not formally published. Required fields: `author`, `title`, `note`. Optional fields: `month`, `year`.

In addition to the fields listed above, each entry type also has an optional `key` field, used in some styles for alphabetizing and forming a `\bibitem` label. You should include a `key` field for any entry with no `author` or author substitute. (Depending on the entry type, an `editor` or an `organization` can substitute for an author.) Do not confuse the `key` field with the key that appears in the `\cite` command and at the beginning of the whole entry, after the entry type.

B.2.2 Fields

Below is a description of all the fields recognized by the standard bibliography styles. An entry can also contain other fields that are ignored by those styles.

address Usually the address of the `publisher` or `institution`. For major publishing houses, omit it entirely or just give the city. For small publishers, you can help the reader by giving the complete address.

annote An annotation. It is not used by the standard bibliography styles, but may be used by other styles that produce an annotated bibliography.

author The name(s) of the author(s), in the format described above.

booktitle The title of a book, a titled part of which is being cited. It is used only for the `incollection` and `inproceedings` entry types; use the `title` field for `book` entries. How to type titles is explained above.

chapter A chapter (or other sectional unit) number.

crossref The database key of the entry being cross-referenced.

edition The edition of a book—for example, "Second". (The style will convert this to "second" if appropriate.)

editor The name(s) of editor(s), typed as indicated above. If there is also an `author` field, then the `editor` field gives the editor of the book or collection in which the reference appears.

howpublished How something strange was published.

institution The sponsoring institution of a technical report.

journal A journal name. Abbreviations may exist; see the *Local Guide.*

key Used for alphabetizing and creating a label when the `author` and `editor` fields are missing. This field should not be confused with the key that appears in the `\cite` command and at the beginning of the entry.

month The month in which the work was published or, for an unpublished work, in which it was written. Use the standard three-letter abbreviations described above.

note Any additional information that can help the reader. The first word should be capitalized.

number The number of a journal, magazine, technical report, or work in a series. An issue of a journal or magazine is usually identified by its volume and number; the organization that issues a technical report usually gives it a number; books in a named series are sometimes numbered.

organization The organization that sponsors a conference or that publishes a manual.

pages One or more page numbers or ranges of numbers, such as `42--111` or `7,41,73--97`.

publisher The publisher's name.

school The name of the school where a thesis was written.

series The name of a series or set of books. When citing an entire book, the `title` field gives its title and the optional `series` field gives the name of a series or multivolume set in which the book was published.

title The work's title, typed as explained above.

type The type of a technical report—for example, "Research Note". It is also used to specify a type of sectional unit in an `inbook` or `incollection` entry and a different type of thesis in a `mastersthesis` or `phdthesis` entry.

volume The volume of a journal or multivolume book.

year The year of publication or, for an unpublished work, the year it was written. It usually consists only of numerals, such as `1984`, but it could also be something like `circa 1066`.

B Sample Pages of the Dissertation

B.1 Title page

Title of the Dissertation

Name of the author

Month Year

Abstract

This is where an abstract can appear. If you don't have an abstract, just throw out theses lines. The abstract should not exceed 10 lines :-)
3
4
5
6
7
8
9
10

Project Dissertation submitted to the Swansea University
in Partial Fulfilment for the Degree of Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

B.2 Declaration page

Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

`<date>`

Signed:

Statement 1

This dissertation is being submitted in partial fulfilment of the requirements for the degree of a BSc in Computer Science.

`<date>`

Signed:

Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

`<date>`

Signed:

Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

`<date>`

Signed:

C Marking sheets

All the project marking schemes are now available online at:
https://cs.swansea.ac.uk/~csmora/project/Marking/2019-2020/MarkingForms.php

D Project deadlines 2019-2020

Teaching		Project deadline	
Induction lecture	26/09/2019		
Project's Initial Document	0?/11/2019		
Teaching starts	30/09/2019		
		Initial document	0?/11/2019
Talk on Gregynog Presentation	28/10/2019		
(dates to be announced - 3 Cohorts)		Gregynog	24-30/11/2019
Christmas Recess/Revision	16/12/2019- 03/01/2020		
Winter Graduation	16/12/2019– 20/12/2019		
Assessment	06/01/2020- 24/01/2020		
Start of second term	27/01/2020		
none			
Talk on Poster presentation			
& Dissertation	31/03/2020		
(date to be announced)		*Recommended:*	last week
		Discuss dissertation outline	before
		with supervisor	Easter recess
Easter recess	06/04/2020– 24/04/2020		
Teaching starts	27/04/2020		
Project dissertation	??/04/2020		
		Project demonstration Fair	??/04/2020
Revision week	04/05/2020– 08/05/2020		
Assessment	11/05/2020– 05/06/2020		

Disclaimer The above dates are based upon the dates of the academic year 2019-2020as published by the University. If circumstances require it, the department might changes any of these dates – of course with due notice. Errors may also happen.