Brunel University London

Software Planning

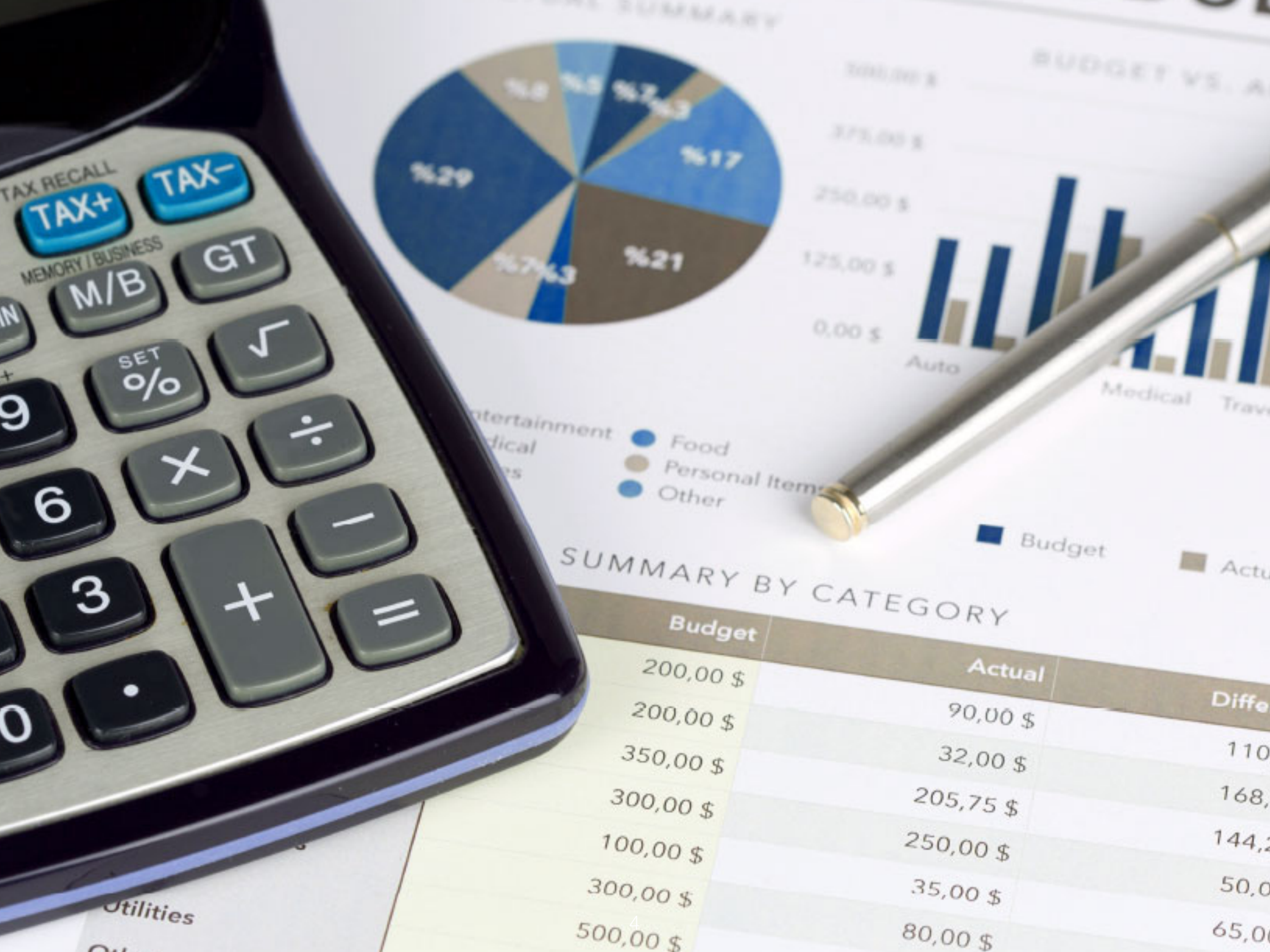
CS2002 Software Development and Management

Giuseppe Destefanis
giuseppe.destefanis@brunel.ac.uk

You have to plan a trip to Amsterdam

Trip information

- 3 nights
- Leaving on Friday
- Back on Monday afternoon the latest


TAX RECALL
TAX+
TAX–
MEMORY / BUSINESS
M/B
GT
SET
%
÷
×
−
+
=
%29
%17
%21
Food
Personal Items
Other
Budget
SUMMARY BY CATEGORY
Budget
Actual
200,00 $
200,00 $
350,00 $
300,00 $
100,00 $
300,00 $
500,00 $
90,00 $
32,00 $
205,75 $
250,00 $
35,00 $
80,00 $
Utilities

Estimation Techniques

Organizations need to make software effort estimates

Experience-based techniques

The estimate of future effort requirements

- Based on a manager's experience of past projects and the application domain.

- Essentially, the manager makes an informed judgment of what the effort requirements are likely to be

Algorithmic cost modelling

- In this approach, a formulaic approach is used to compute the project effort
- Based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved

Experience-based techniques

- It relies on judgments based on experience of past projects and effort expended in these projects
  - NASA
- Typically, you:
  - Identify the deliverables to be produced in a project and the different software components or systems that are to be developed
  - Document these in a spreadsheet, estimate them individually and compute the total effort required
- It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate

Experience-based problems

- A new software project may not have much in common with previous projects
    - Analogy-based studies
- Experience relies on staff being available/still around with the experience

Algorithmic cost modelling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:
  - **Effort = 25 x Size^B x M**
    - 25 is an organisation-recognised constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.
- The most commonly used product attribute for cost estimation is code size
  - But this is counter-intuitive
- Most models are similar but they use different values for B and M

Algorithmic estimation accuracy

- The size of a software system can only be known accurately when it is finished

- So you’re trying to predict the future!

Algorithmic estimation accuracy

- Several factors influence the final size
  - Use of reused systems and components
  - Programming language used
- As the development process progresses
  - The size estimate becomes more accurate
- The estimates of the factors contributing to B and M are subjective
  - Vary according to the judgment of the estimator

Top-down and bottom-up estimation

- Either of these approaches may be used top-down or bottom-up.
- **Top-down**
  - Start at the system level and assess the overall system functionality and how this is delivered through sub-systems.
- **Bottom-up**
  - Start at the component level and estimate the effort required for each component.
  - Add these efforts to reach a final estimate.

Lines of Code

```
class LoggingHandler<T>   : Handles<T> where T:Command
{
    private readonly Handles<T> next;

    public LoggingHandler(Handles<T> next)
    {
        this.next = next;
    }

    public void Handle(T command)
    {
        myLoggingFramework.Log(command);
        next.Handle(command);
    }
}


 var handler = new LoggingHandler<DeactivateCommand>(
                    new DeactivateCommandHandler(...)
                );
```

Lines of Code

- What's a line of code?
    - The measure was first proposed when programs were typed on cards with one line per card
    - How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line.
        - *Empty space*
        - *Comment lines*
        - *}*
- Getters and setters

Lines of Code

- What programs should be counted?
  - One system we looked at had had no defects for 12 months
  - Then we were told why
- Number of correct lines of code a developer produces each day?

Productivity Comparison

The lower level the language, the more productive in terms of code size the programmer, because:

- The same functionality takes more code to implement in a lower-level language than in a high-level language
- The more verbose the programmer, the higher the productivity

- Measures of productivity based on lines of code suggest that programmers who write verbose code are more productive than programmers who write compact code

Factors affecting productivity

Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced. Brook's Law.
Technology support	Good support technology such as decent tools, etc. can improve productivity.
Working environment	A quiet working environment with private work areas contributes to improved productivity.

PeerJ Computer Science

Software development: do good manners matter?

Giuseppe Destefanis[1], Marco Ortu[2], Steve Counsell[1], Stephen Swift[1], Michele Marchesi[2] and Roberto Tonelli[2]

[1] Department of Computer Science, Brunel University, London, United Kingdom
[2] Department of Electrical and Electronic Engineering, University of Cagliari, Cagliari, Italy

ABSTRACT

A successful software project is the result of a complex process involving, above all, people. Developers are the key factors for the success of a software development process, not merely as executors of tasks, but as protagonists and core of the whole development process. This paper investigates social aspects among developers working on software projects developed with the support of Agile tools. We studied 22 open-source software projects developed using the Agile board of the JIRA repository. All comments committed by developers involved in the projects were analyzed and we explored whether the politeness of comments affected the number of developers involved and the time required to fix any given issue. Our results showed that the level of politeness in the communication process among developers does have an effect on the time required to fix issues and, in the majority of the analysed projects, it had a positive correlation with attractiveness of the project to both active and potential developers. The more polite developers were, the less time it took to fix an issue.

Subjects Data Mining and Machine Learning, Data Science, Software Engineering

Keywords Social and human aspects, Politeness, Mining software repositories, Issue fixing time, Software development

Are Bullies more Productive? Empirical Study of Affectiveness vs. Issue Fixing Time

Marco Ortu*, Bram Adams ‡, Giuseppe Destefanis†, Parastou Tourani ‡, Michele Marchesi * Roberto Tonelli *
*DIEE, University of Cagliari, Italy, {marco.ortu,michele,roberto.tonelli}@diee.unica.it
†CRIM, Computer Research Institute of Montreal, Canada, {giuseppe.destefanis}@crim.ca
‡École Polytechnique de Montréal, Canada, {bram.adams,parastou.tourani}@polymtl.ca

***Abstract*—Human Affectiveness, i.e., the emotional state of a person, plays a crucial role in many domains where it can make or break a team's ability to produce successful products. Software development is a collaborative activity as well, yet there is little information on how affectiveness impacts software productivity. As a first measure of this impact, this paper analyzes the relation between sentiment, emotions and politeness of developers in more than 560K Jira comments with the time to fix a Jira issue. We found that the happier developers are (expressing emotions such as *JOY* and *LOVE* in their comments), the shorter the issue fixing time is likely to be. In contrast, negative emotions such as *SADNESS*, are linked with longer issue fixing time. Politeness plays a more complex role and we empirically analyze its impact on developers' productivity.**

***Index Terms*—Affective Analysis, Issue Report, Empirical Software Engineering**

I. Introduction

Team sports like soccer [1] are a primary example that the productivity of an organization is not only a product of the talent in a team, but depends heavily on human affectiveness, i.e., the way in which individuals feel and how they perceive their colleagues [2]. A rude coach without people management skills will only alienate his team, prompting them to just do anything to avoid his scorn rather than focusing on winning the

for example in exchanges between the creator of the Linux kernel and some of the Linux developers[1].

In previous research [9], the authors manually analyzed whether discussion boards like bug repositories contain emotional content. They indeed found evidence of gratitude, joy and sadness, and also weak evidence that the presence of emotions like gratitude was related with faster issue resolution time. However, due to the manual nature of the analysis, the data sample was relatively limited. Furthermore, emotions are but one of the possible human affectiveness measures, and might not have the strongest relation with issue resolution time.

In this paper, we empirically analyze more than 560K comments of the Apache projects' Jira issue tracking system to understand the relation between human affectiveness and developer productivity. In particular, we extract affectiveness metrics for emotion, sentiment and politeness, then build regression models to understand whether these metrics can explain the time to fix an issue. We aim to address the following research questions:

RQ1: *Are emotions, sentiment and politeness correlated to each other?*

Designing for security

- Design guidelines encapsulate good practice in secure systems design
- Design guidelines serve two purposes:
  - *They raise awareness of security issues in a software engineering team*
    - *Security is considered when design decisions are made*
    - *They can be used as the basis of a review checklist that is applied during the system validation process*
- Design guidelines here are applicable during software specification and design
- Cybersecurity
- The FindBugs tool

Design guidelines for secure systems engineering

Security guidelines	
Base security decisions on an explicit security	
Avoid a single point of failure	
Fail securely	
Balance security and usability	
Log user actions	
Use redundancy and diversity to reduce risk	
Specify the format of all system inputs	
Compartmentalize your assets	
Design for recoverability	

Design guidelines 1-3

- **Base decisions on an explicit security policy**
    - Define a security policy for the organization that sets out the fundamental security requirements that should apply to all organizational systems
        - USBs
- **Avoid a single point of failure**
    - Ensure that a security failure can only result when there is more than one failure in security procedures.
    - For example, have password and question-based authentication
- **Fail securely**
    - When systems fail, for whatever reason, ensure that sensitive information cannot be accessed by unauthorized users

Design guidelines 4-6

- **Balance security and usability**
  - Try to avoid security procedures that make the system difficult to use
  - Sometimes you have to accept weaker security to make the system more usable
- **Log user actions**
  - Maintain a log of user actions that can be analyzed to discover who did what.
  - If users know about such a log, they are less likely to behave in an irresponsible way
- **Use redundancy and diversity to reduce risk**
  - Keep multiple copies of data and use diverse infrastructure so that an infrastructure vulnerability cannot be the single point of failure

Design guidelines 7-9

- **Specify the format of all system inputs**
    - If input formats are known then you can check that all inputs are within range so that unexpected inputs don’t cause problems
- **Compartmentalize your assets**
    - Organize the system so that assets are in separate areas
    - Users only have access to the information that they need rather than all system information
- **Design for recoverability**
    - Design the system to simplify recoverability after a successful attack (‘graceful degradation’)

Reading

- Somerville, Chapters 13, 23 and 24