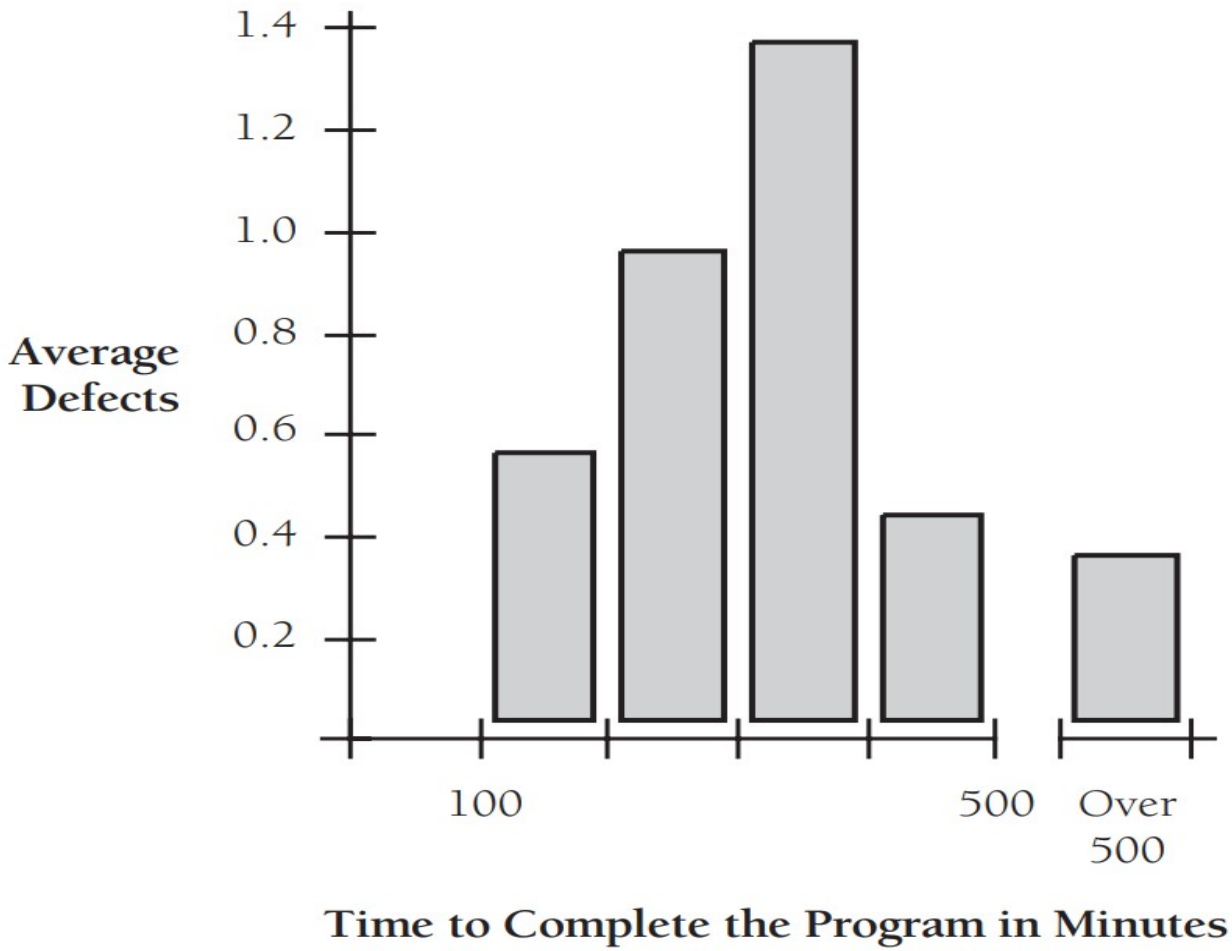
# Software Engineering CS3003

Lecture 8: Clean code




De Marco and Lister (1985)

# Structure of lecture

- What is clean code?
- The rules for clean code
  - Examples:
    - Naming conventions
    - Rules for writing comments
- The role of legacy systems in a clean code approach
  - Legacy system components
- Some “team” theory

# Lecture schedule

| Week | Lecture Topic | Lecturer | Week Commencing |
|---|---|---|---|
| 1 | Introducing the module and Software Engineering | Steve Counsell | 20th Sept. |
| 2 | Software maintenance and Evolution | Steve Counsell | 27th Sept. |
| 3 | Software metrics | Steve Counsell | 4th Oct. |
| 4 | Software structure, refactoring and code smells | Steve Counsell | 11th Oct. |
| 5 | Test-driven development | Giuseppe Destefanis | 18th Oct. |
| 6 | Software complexity<br>**Coursework released Tues 26th Oct.** | Steve Counsell | 25th Oct. |
| 7 | **ASK week** | **N/A** | **1st Nov** |
| 8 | Software fault-proneness | Steve Counsell | 8th Nov. |
| 9 | Clean code | Steve Counsell | 15th Nov. |
| 10 | Human factors in software engineering | Giuseppe Destefanis | 22th Nov. |
| 11 | SE techniques applied in action | Steve Counsell | 29th Nov. |
| 12 | Guest Lecture (tba)<br>**Coursework hand-in 6th December** | Guest Lecture | 6th Dec. |

# Lab schedule

| Week | Labs | Week Commencing |
|---|---|---|
| 1 | **No labs** | 20th Sept. |
| 2 | Lab (Introduction) | 27th Sept. |
| 3 | Lab | 4th Oct. |
| 4 | Lab | 11th Oct. |
| 5 | Lab | 18th Oct. |
| 6 | **No lab** | 25th Oct. |
| 7 | **ASK week** | **1st Nov.** |
| 8 | Lab | 8th Nov. |
| 9 | Catch-up Lab | 15th Nov. |
| 10 | Work on coursework (no Lab) | 22nd Nov. |
| 11 | Work on coursework (no Lab) | 29th Nov. |
| 12 | **No lab** | 6th Dec. |

# What is clean code?

- Code that is elegant
  - Should be pleasing to read
- Code that is focused
  - Each function, each class exposes a single-minded attitude that remains entirely un-distracted by the surrounding details
- Code that is (and has been) taken care of by its curators
  - The developers have kept it simple and orderly
- Code that contains no duplication
  - The number of entities such as classes and methods is minimised

# What is clean code?

*“I like my code to be elegant and efficient.”*

*“The logic should be straightforward to make it hard for bugs to hide.”*

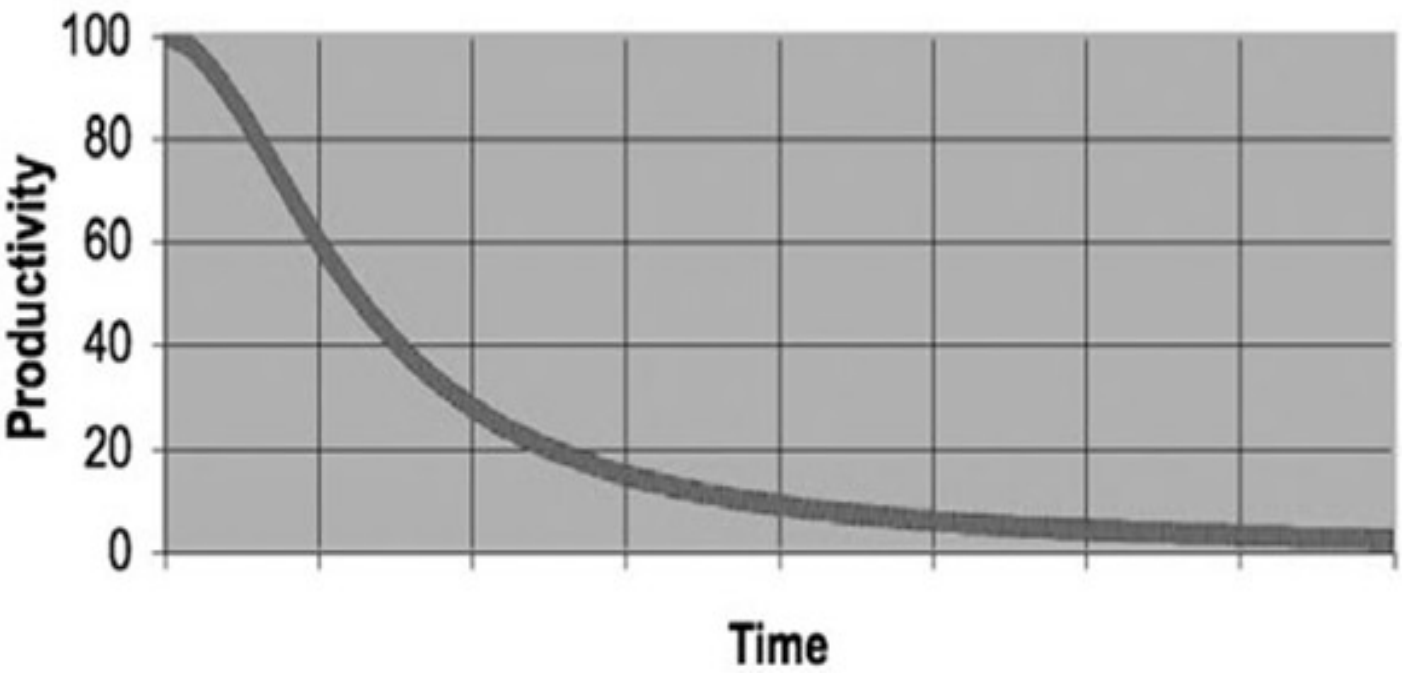
*“Clean code does one thing well.”*

**Bjarne Stroustrup**, inventor of C++

# Why bother with clean code?

- Clean code means better use of time in the long-term
  - Short term pain for long term gain
  - Saves money
- Clean code means easier ramp-up of new staff
  - Ramp-up refers to getting new team members up to speed on the system
- Clean code means easier debugging
- Clean code means easier maintenance in the long term
  - Smells, decay, bathtub, Lehman's Laws
- Clean code means fault reduction

# Consequences of unclean code



# General rules for clean code

- Follow “common sense” principles
  - Keep it simple stupid (KISS)
    - Simpler is always better (reduce complexity as much as possible)
    - Refactoring
  - Don’t let technical debt accrue
  - You Aren't Gonna Need It (YAGNI)
    - A developer should not add functionality unless deemed necessary

# Naming

# Use Meaningful names

*Bad*

```
int d; // elapsed time in days
```

*Good*

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

# Use intention revealing names

*Bad*

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
      if (x[0] == 4)
        list1.add(x);
    return list1;
}
```

*Good*

```
public final static int STATUS_VALUE = 0;
public final static int FLAGGED = 4;

public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
      if (cell[STATUS_VALUE] == FLAGGED)
        flaggedCells.add(cell);
    return flaggedCells;
}
```

# Use pronounceable names

*Bad*

```
class DtaRcrd102 {
   private Date genymdhms;
   private Date modymdhms;
   private final String pszqint = "102";
};
```

*Good*

```
class Customer {
   private Date generationTimestamp;
   private Date modificationTimestamp;
   private final String recordId = "102";
};
```

# Use “searchable” names

- Use names that can be searched for easily in the code
  - Avoid single letter names for variables (x, y, etc)

# Variable length

**Table 11-2 Variable Names That Are Too Long, Too Short, or Just Right**

| | |
|---|---|
| Too long: | *numberOfPeopleOnTheUsOlympicTeam* |
| | *numberOfSeatsInTheStadium* |
| | *maximumNumberOfPointsInModernOlympics* |
| Too short: | *n, np, ntm* |
| | *n, ns, nsisd* |
| | *m, mp, max, points* |
| Just right: | *numTeamMembers, teamMemberCount* |
| | *numSeatsInStadium, seatCount* |
| | *teamPointsMax, pointsRecord* |

Code Complete, Second Edition, McConnell

# Java “accepted” conventions

## Java Conventions

In contrast with C and C++, Java style conventions have been well established since the language's beginning:

- *i* and *j* are integer indexes.
- Constants are in *ALL_CAPS* separated by underscores.
- Class and interface names capitalize the first letter of each word, including the first word—for example, *ClassOrInterfaceName*.
- Variable and method names use lowercase for the first word, with the first letter of each following word capitalized—for example, *variableOrRoutineName*.
- The underscore is not used as a separator within names except for names in all caps.
- The *get* and *set* prefixes are used for accessor methods.

Code Complete, Second Edition, McConnell

# Class and method namings

- Always write your classes and methods on the assumption that the next person who will read your code is a very angry and fastidious person and they know who you are
  - Class and method names should convey what the class and methods do:
    - Class names like Person, Car, Record
  - Methods should try to make method names a “doing” thing:
    - Get, set, amend etc

# Methods

- Methods should minimise the number of parameters they have
- Method callers and callees should be close
  - If one method calls another:
    - Keep those methods vertically close in the source file
    - Ideally, keep the caller right above the callee
  - We tend to read code from top-to-bottom, like a newspaper
    - Because of this, make your code read that way

# Use the same concepts

- Use the same concept across the codebase.
  - FetchValue() vs GetValue() vs RetrieveValue()
- If you are using fetchValue() to return a value of something,
  - Always use fetchValue()
    - Do NOT use getValue() or retrieveValue() to mean the same thing
    - Otherwise, you will confuse the reader

# Use opposites properly

- Add/remove
- Start/stop
- Begin/end
- Show/hide
- Min/max
- Insert/delete
- Open/close
- Get/put

# Comments

*“Don’t comment bad code—rewrite it.”*

—Brian W. Kernighan and P. J. Plaugher

# Comments

- Can be helpful or damaging
  - Poor commenting is worse than no commenting
- Inaccurate comments are worse than no comments at all
- Used to compensate failure expressing with code
  - Refactor instead
- They often lie
  - Because they are out of date
- Must have them, but minimize them

# Comments – FIVE guidelines

- Don’t comment poor code, rewrite the code
- Do warn of consequences in the code
  - Example: “this code will take a while to run”
- Do emphasize important points in the code
  - Example: “sometimes the data input has leading space”
- Don’t include “noise” in you comments
  - Example: “// this is declaration of an int called “day”
    - int day;
- Don’t copy and paste comments, because you’ll always forget to make the necessary changes

# Comments (cont.)

Express yourself in code!!

*Bad*

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

*Good*

```
if (employee.isEligibleForFullBenefits())
```

*“The most reliable document of software is the code itself. In many cases, the code is the only documentation. Therefore, strive to make your code self-documenting, and where you can’t, add comments*.”

*Daniel Read:*
The Principle of Self-Documenting Code

# Code Style Issues

- Keep code lines short
- Use appropriate indentation
- Declare variables close to where they are used
- Use vertical blank space sensibly
  - To associate related things and disassociate unrelated things
- Keep horizontal blank space consistent

# What is “Dirty” code

- It is rigid
  - The software is difficult to change
  - A single change causes a cascade of other required changes
- It is fragile
  - The software breaks in many places after a single change
- It is immobile
  - You cannot reuse parts of the code because of high effort and risks in doing so

# Legacy systems

# What are legacy systems?

- Software systems that are developed especially for an organisation have a long lifetime
- Many software systems that are still in use were developed many years ago
  - Using technologies that are now “old”
- These systems are still business critical
  - Essential for the normal functioning of the business

# Changing legacy systems

- Changing legacy systems is often expensive
  - Different parts implemented by different teams
    - so there is no consistent programming style
  - The system uses an old language (e.g. COBOL)
  - The system documentation is often out-of-date
  - The system structure may be corrupted by many years of maintenance (bathtub curve)
  - Expertise may no longer exist in specific areas

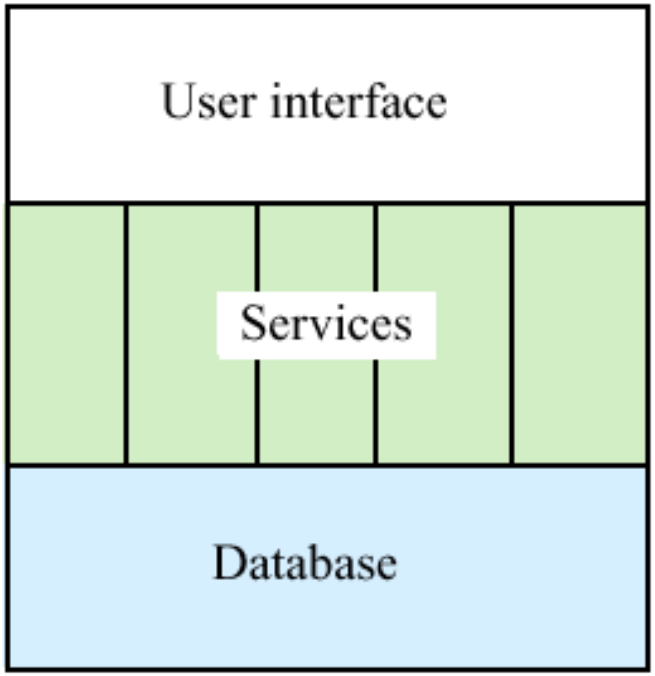
# The dilemma of legacy systems

- It is expensive and risky to replace the legacy system AND…..
- It is expensive to maintain the legacy system
- Businesses must weigh up the costs and risks and may choose to extend the system lifetime using techniques such as re-engineering
- See the “audit grid” from an earlier lecture
  - Helps to make decisions

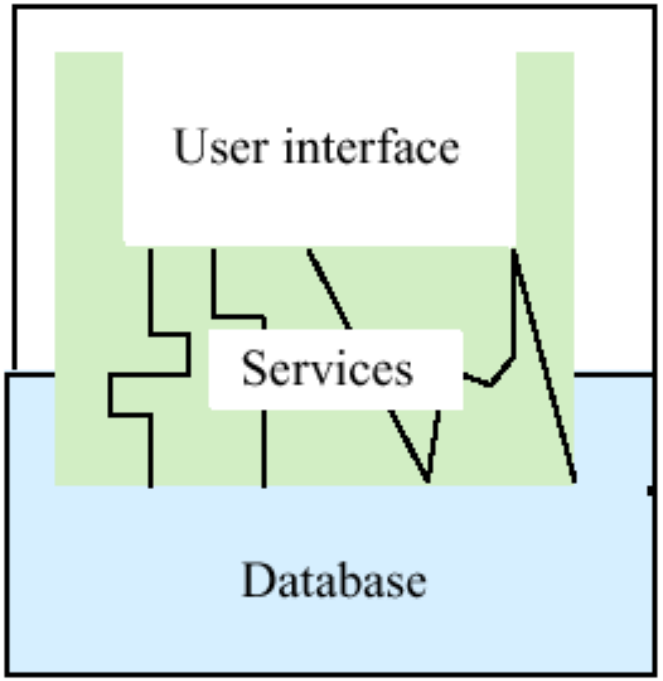
# Legacy system structures

- Legacy systems can be considered to be socio-technical systems and not simply software systems
  - System hardware - may be mainframe hardware
  - Support software - operating systems and utilities
  - Application software - several different programs
  - Application data - data used by these programs that is often critical business information
  - Business processes - the processes that support a business objective and which rely on the legacy software and hardware
  - Business policies and rules - constraints on business operations

# Legacy system structures

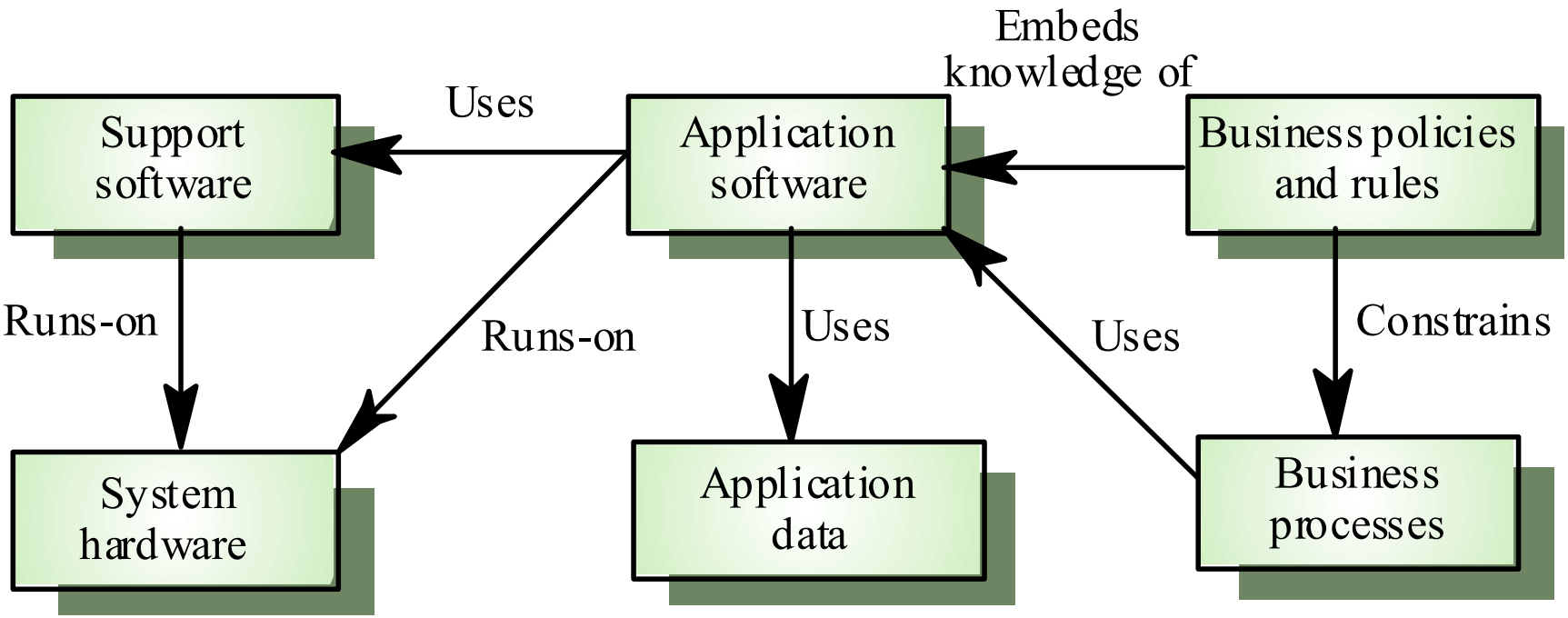



Ideal model for distribution




Real legacy systems

# Legacy system components




Close
LG
Source: http://www.quora.com/How-will-the-world-end

# Team theory

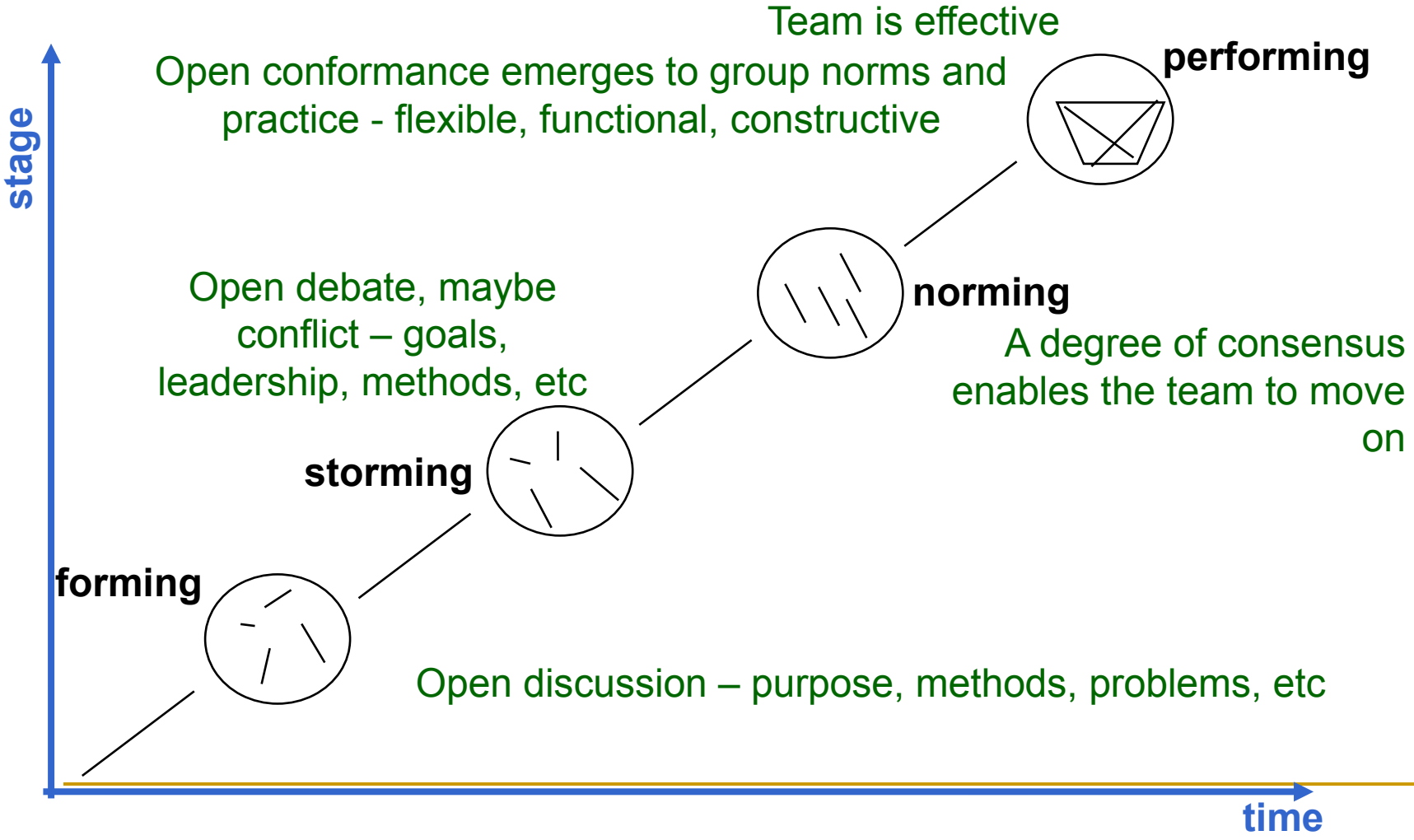
# Tuckman's theory of teams

- Developed by Bruce Tuckman in the 1960's
- As valid now for the agile approach as it was then
- Suggests that every team goes through four key stages and that:
  - "*Phases are all necessary and inevitable in order for a team to grow, face up to challenges, tackle problems, find solutions, plan work, and deliver results*."

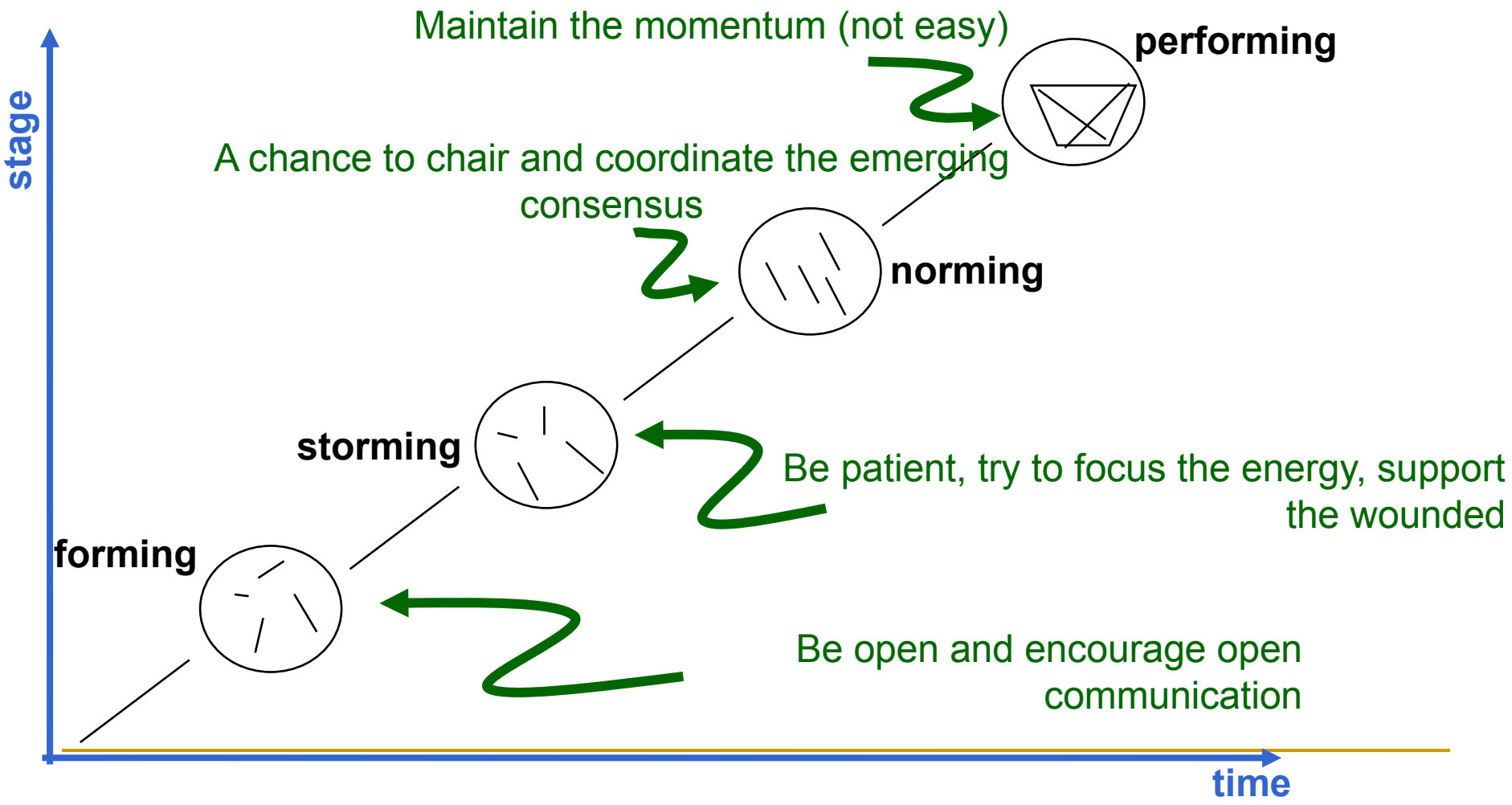
# Bruce Tuckman (US psychologist)

extensionaus.com.au

# Tuckman theory of the team



# What is the Project Manager's role in all this?



# Personal characteristics of software engineers (McConnell)

- Characteristics that seem to matter most are:
  - Humility:
    - Know your weaknesses and be honest about them to others
  - Curiosity:
    - Take the trouble to learn about successful projects
    - This is what NASA has done for decades
  - Intellectual honesty:
    - Admit all mistakes and try not to hide those mistakes
  - Justifiable laziness:
    - Find a quick way to solve a problem so that you spend as least time sorting out that problem
      - This is very different to ignoring a problem through laziness

# Questions

- Question 1: “Legacy systems are trivial to keep up-to-date and their maintenance is also trivial”. Discuss this statement.
- Question 2: What practices would you consider sensible for naming your *variables* when you write code? Give some examples to illustrate. What good practice would you also use when you write functions (i.e., OO classes)?
- Question 3: With the aid of a diagram, describe the components of a legacy system.
- Question 4: What would your advice be on the writing of comments in code to ensure that you make the code as maintainable as possible?

# Coursework

- Don’t forget to number your sections
- Don’t forget to name the two systems you chose and the three metrics
- Don’t forget to label your diagrams
- Don’t forget to put in at least five references
- Don’t forget to stick to the word count guidance as much as you can

# Reading for the week

- Robert C. Martin, *"Clean code: A Handbook of Agile Software Craftsmanship"*
- Some decent links:
  - https://simpleprogrammer.com/clean-code-principles-better-programmer/
  - https://blog.cleancoder.com/
  - https://www.planetgeek.ch/wp-content/uploads/2013/06/Clean-Code-
  - http://campus.murraystate.edu/academic/faculty/wlyle/430/CleanCode.htm
  - https://stevemcconnell.com/articles/

# Reading for the week (cont.)

- DeMarco, Tom, and Timothy Lister. 1985. Programmer Performance and the Effects of the Workplace. Proceedings of the 8th International Conference on Software Engineering: 268–272
- https://dzone.com/articles/naming-conventions-from-uncle-bobs-clean-code-phil