

Introduction to Software Development:
The Software Development Process
Alex Poulouvassilis

Outline of this lecture

1. The Software Development Process
 - phases involved
 - features of “good” software design
 - algorithm design, pseudocode, types of algorithms
2. Software Development Methodologies
 - waterfall, incremental, prototyping, agile
3. Deriving Designs

1. The Software Development Process

- Software development is usually carried out by several people; on this course it is being carried out by just one, though – you!
- The different roles involved in software development include ***business analyst, systems analyst, project manager, developer, Quality Assurance*** (QA)
- On ISD module we are focussing mainly on development and QA
- The Information Systems module discusses the other roles; it also introduces the ***Unified Modeling Language*** (UML) which aims to provide modelling techniques for defining the artefacts produced during most phases of software development
- In the IS module, UML is used for the ***requirements, specification*** and ***system design*** phases of software development (see below)
- In the OODP module, UML is used for more detailed analysis design of object-oriented programs

Software Development Phases

- Software development consists of a number of ***phases*** (see below)
- The ordering and repetition of the different phases depends on the particular ***software development methodology*** which is being followed in order to develop a given software system (see later)
- Different methodologies are better suited to different circumstances (see later)
- The major phases in software development are: Feasibility study; Requirements Analysis; Specification; System Design; Detailed Design; Implementation & Unit Testing; System Testing; and Deployment, Operation & Maintenance
- We now discuss these phases in more detail

(i) Feasibility study

- This phase is the responsibility of the person commissioning a new system or an upgrade to an existing system
- The objectives of this phase include :
 - investigation of existing practices in the area to be addressed by the new system
 - estimation of the cost and benefit of replacing them by the new system (cost/benefit analysis)
 - estimation of whether the new system can be produced as required, within the available time and budget, and using the available technical and personnel resources (risk analysis)

(ii) Requirements Analysis

- This phase is undertaken by the business and systems analysts, and its objectives include
 - identifying the main tasks that will be addressed by the system – these are known as the ***functional requirements***
 - determining how the system will interface with existing processes, both automated and manual processes
 - identifying ***non-functional requirements*** such as performance, security and reliability requirements of the proposed system

(iii) Specification

- The objectives of this phase are to:
 - describe how the proposed system will satisfy the functional and non-functional requirements
 - provide a precise description of the software to be developed (the “spec”)
 - address Quality Assurance i.e. what procedures will be put in place by the project team to manage the development and testing of the system in order to ensure a high quality product that satisfies the requirements

Specification (cont'd)

- The Unified Modelling Language (UML) provides a broad range of modelling techniques for representing functional requirements e.g. to produce
 - Use cases
 - Domain models
 - Class diagrams (static view of the software)
 - Interaction diagrams (dynamic view of the software)
- User interfaces and reports layouts are typically specified at the specification stage
- Details of the above are discussed in the OODP and Information Systems modules. For the purposes of this ISD module we will assume a simple English textual description of the “spec”
- Also specified at this stage are the formats of files and databases that will support the data ***persistence*** requirements of the system – discussed later on in this ISD module

(iv) System Design

- The objective of this phase is to design a system which satisfies the “spec”, aiming to achieve the features of a “good” design (see below)
- One aspect of System Design is a more detailed ***data analysis*** than occurred in phase (iii), for example leading to more elaborated class diagrams (see OODP module) and to logical and physical database designs (see DM and DKE modules)
- Another aspect of System Design is a more detailed ***functional analysis*** than occurred in phase (iii), leading to a more detailed structuring of the system into “program units” e.g. classes, methods, functions ...

What is a “good” design?

- Like any creative process, software design is subjective and there is not usually a single correct design for a system
- However, we do need criteria to evaluate alternative designs
- Peter Coad and Edward Yourdon in their seminal book “Object-Oriented Design”, Prentice-Hall, 1991, state that:
“a good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime”
- The cost of a system over its entire lifetime is made up of costs associated with analysis, design, programming, testing, operation and maintenance. Of these, maintenance costs are often the largest component
- Thus, a design that leads to an easily maintainable system can be regarded as being a good design

How can we achieve a “good” design?

- Four key design techniques have been found to improve the maintainability of software: ***abstraction, information hiding, high cohesion*** and ***low coupling***.
- **Abstraction:**
 - this means defining program units that can be used in ways that allow their implementation details to be ignored
 - this allows the developer to focus their attention at the necessary level of detail when designing new program units that use existing ones
 - there are two types of abstraction: ***procedural abstraction*** and ***data abstraction***

Procedural abstraction

- this identifies the main *processing* that needs to be carried out by the system (or part of the system) and repeatedly decomposes this processing into sub-processing
- at some point, the identification of individual program units occurs and the contents of one program unit reflect one part of the overall processing carried out by the system
- Groovy methods that call other methods to undertake some of their sub-processing are program units that exemplify procedural abstraction

Data abstraction

- this identifies the main *classes of objects* to be manipulated by the system, together with their attributes and operations, and breaks them down into increasingly finer-grained object classes with their attributes and operations
- at some point the identification of individual program units occurs and the contents of one unit reflect one class of objects, with their attributes and operations
- Groovy classes are program units that exemplify data abstraction
- A class represents a set of objects, their attributes (fields) and the operations to be performed on them (constructors and methods)
- Groovy classes can be structured into hierarchies of sub- and super-classes

Information hiding

- **Information hiding:**
 - this means that the information within each program unit should only be accessible by other program units if they need to use it in order to undertake their intended task
 - in this way, the “hidden” content of a program unit is protected from errors that may inadvertently be introduced elsewhere in the software
 - Examples of information hiding in Groovy are
 - private fields, constructors, methods and classes, that cannot be accessed outside the class
 - variables declared inside classes
 - variables declared inside methods

High cohesion

- **High cohesion:**
 - Cohesion is the degree to which the contents of a program unit contribute to implementing a single well-defined task
 - If the individual units have high cohesion, then the overall system is more easily understandable, testable and reusable
 - Also, a change in the system's requirements can be localized to just a few units, aiding the maintainability of the system
- For example, in order to exhibit high cohesion,
 - Groovy classes should contain information about a set of similar objects, with similar attributes and operations; so that change in the requirements for that set of objects can be addressed by modifying the corresponding class
 - methods in Groovy should undertake one specific operation; so that a change in the requirements for that operation can be addressed by modifying the corresponding method

Low coupling

- **Low coupling:**

- Coupling reflects how many dependencies there are between program units
- If changes in one unit do not make it necessary to change also many other units, then this indicates low, or “loose”, coupling
- Low coupling improves the maintainability of a system since it reduces the likelihood of programming errors being introduced when part of a system is changed

- Ways to reduce coupling:

- avoiding `public` fields in classes
- making sure that each class is responsible for handling its own data (storing it, manipulating it, accessing it), and that it provides appropriate `public` methods for other classes that need to access this data to use

(v) Detailed Design

- This occurs at the level of individual program units. The objective is to produce unit designs which meet their specification, striving for the features of a “good” design
- **Algorithm design** is a major aspect of detailed design:
 - an **algorithm** is a “recipe” or “set of instructions”, which can subsequently be translated into a particular programming language to be run on a computer
- We can define algorithms using a structured form of English, called **pseudocode**, which is independent of any particular programming language

Pseudocode

- Pseudocode is a structured form of English that is useful for defining algorithms (the word comes from the Greek “pseudo” meaning “false” and “codex” meaning “code”)
- Why use pseudocode?
 - It allows us to focus designing how algorithm will work without having to worry about the syntactic details of implementing it in a particular programming language
 - We can stop at just the design e.g. in algorithms research
- Pseudocode supports:
 - sequencing of instructions
 - repetition of a group of instructions (`for` and `while` loops)
 - calling of sub-algorithms
 - following different branches of the “recipe” (`if ... then ... else ...`)

Types of Algorithms

- Algorithms that use a similar problem-solving approach can be grouped together
- Some types of algorithms are:
 - Simple recursive algorithms
 - Divide and conquer algorithms
 - Dynamic programming algorithms
 - Backtracking algorithms
- We will look at examples of each of these next

Simple Recursive Algorithms

- These algorithms
 - (i) solve the base case (or base cases) directly; and
 - (ii) call themselves to solve a simpler sub-problem, and do some more work in order to convert the solution to the simpler sub-problem into a solution to the overall problem
- These are called “simple recursive algorithms” because several other algorithm types are also inherently recursive
- Following are some examples of simple recursive algorithms:
 - Computing the factorial of a non-negative integer n
 - Computing the length of list
 - Checking if a value occurs in a list

Computing the factorial of a non-negative integer n

Algorithm `factorial`

Input: non-negative integer n

Output: positive integer n

```
if n == 0
then return 1
else return n * (factorial (n-1))
```

E.g.

- $\text{factorial}(0) \rightarrow 1$
- $\text{factorial}(1) \rightarrow 1 * \text{factorial}(0) \rightarrow 1 * 1 \rightarrow 1$
- $\text{factorial}(2) \rightarrow 2 * \text{factorial}(1) \rightarrow 2 * (1 * \text{factorial}(0))$
 $\rightarrow 2 * (1 * 1) \rightarrow 2$

Computing the length of list

Algorithm `length`

Input: a list `L`

Output: a non-negative integer

```
if L is empty
then return 0
else {
  TL = tail of L; /*ie. L minus its first element*/
  return 1 + length(TL)
}
```

E.g.

$$\begin{aligned} \text{length}([2,5,3]) &\rightarrow 1 + \text{length}([5,3]) \rightarrow 1 + (1 + \text{length}([3])) \\ &\rightarrow 1 + (1 + (1 + \text{length}([]))) \\ &\rightarrow 1 + (1 + (1 + 0)) \rightarrow 3 \end{aligned}$$

Computing the length of a list

- How many calls to `length` are there for a list of length N ?
- We say that the algorithm `length` has ***time complexity*** of order N , which we write $O(N)$

Checking if a value occurs in a list

Algorithm `member`

Input: a list `L` and a value `x`

Output: `True` or `False`

```
if L is empty
then return False
else {
    firstL = first element of L;
    TL = tail of L;
    if x == firstL
    then return True
    else return (member(TL, x))
}
```

Checking if a value occurs in a list

- For example:

member ([],5) → False

member([2,5,7,9],7)

→ member([5,7,9],7)

→ member([7,9],7)

→ True

member([2,5,7,9],11)

→ member([5,7,9],11)

→ member([7,9],11)

→ member ([9],11)

→ member ([],11)

→ False

Testing if a value occurs in a list

- How many calls to `member` are there for a list of length N ?
- We say that the algorithm `member` has ***worst case time complexity*** $O(N)$

Exercise for you

Write a simple recursive algorithm to compute the sum of a list of numbers. The input to the algorithm is a list of numbers, L . The output is the sum of these numbers. For example, for the input $L = [3,4,5]$ the output should be 12. If L is empty, the output should be 0.

Test that your algorithm works correctly – show your working.

Computing the sum of a list of numbers

Algorithm `sum`

Input: a list of numbers, `L`

Output: the sum of these numbers

```
if L is empty
then return 0
else {
    firstL = first element of L;
    TL = tail of L;
    return firstL + sum(TL)
}
```

Computing the sum of a list of numbers

Testing the algorithm (try out boundary cases and typical cases):

$$\text{sum}([]) \rightarrow 0$$

$$\text{sum}([9]) \rightarrow 9 + \text{sum}([]) \rightarrow 9 + 0 \rightarrow 9$$

$$\text{sum}([2,5,3])$$

$$\rightarrow 2 + \text{sum}([5,3])$$

$$\rightarrow 2 + (5 + \text{sum}([3]))$$

$$\rightarrow 2 + (5 + (3 + \text{sum}([])))$$

$$\rightarrow 2 + (5 + (3 + 0)) = 10$$

Note on testing

- Note that we are “testing” this pseudocode without actually running it on a computer!
- This is called testing by “manual inspection”, or by “manual walkthrough”
- It is good training, also, for inspecting executable program code without actually running it
- This can be beneficial because (provided your code is well-designed and well-documented!) it allows you to focus on understanding and possibly correcting your code in a different mind set
- This can allow improvements and corrections to be identified that you wouldn’t have thought of in the normal design/implement/test/debug cycle
- It can also be beneficial to show your code to someone else (who may know nothing about it beforehand) for them to manually walk through it, and they may possibly identify improvements or corrections that you might not have spotted for yourself

Divide and Conquer Algorithms

- This type of algorithm divides the problem into smaller sub-problems of the same type, solves these sub-problems recursively, and combines the solutions into a solution for the original problem
- Traditionally, an algorithm is called “divide and conquer” if it contains at least two recursive calls (otherwise it is a simple recursive algorithm)
- Examples of divide and conquer algorithms are computing the n^{th} fibonacci number and sorting a list of numbers using quicksort

Computing the n^{th} Fibonacci number

Algorithm `fib`

Input: positive n

Output: positive integer

```
if (n == 1) or (n == 2)
then return 1
else return (fib(n-1) + fib(n-2))
```

E.g.

$\text{fib}(3) \rightarrow \text{fib}(2) + \text{fib}(1) \rightarrow 1+1 \rightarrow 2$

$\text{fib}(4) \rightarrow \text{fib}(3) + \text{fib}(2) \rightarrow (\text{fib}(2)+\text{fib}(1))+1 \rightarrow 1+1+1 \rightarrow 3$

$\text{fib}(5) \rightarrow \text{fib}(4) + \text{fib}(3) \rightarrow (\text{fib}(3)+\text{fib}(2))+(\text{fib}(2)+\text{fib}(1)) \rightarrow$
 $(\text{fib}(2)+\text{fib}(1))+1+1+1 \rightarrow 1+1+1+1+1 \rightarrow 5$

Sorting a list of numbers – quicksort algorithm

Algorithm `quicksort`

Input: a list of numbers, `L`

Output: `L` sorted according to ordering `<`

`if L is empty or L contains just one element`

`then return L`

`else {`

`firstL = first element of L;`

`TL = tail of L;`

`lesserL = quicksort([x|x in TL; x <= firstL]);`

`greaterL = quicksort([x|x in TL; x > firstL]);`

`sortedL = concatenate(lesserL, [firstL], greaterL);`

`return sortedL`

`}`

quicksort algorithm

E.g. quicksort ([5,2,5,9,1,7])

→ concatenate (quicksort ([2,5,1]), [5], quicksort ([9,7]))

→ concatenate (

 concatenate (quicksort ([1]), [2], quicksort ([5])),
 [5],

 concatenate (quicksort ([7]), [9], quicksort ([])))

→ concatenate (

 concatenate ([1], [2], [5]),
 [5],

 concatenate ([7], [9], []))

→ concatenate ([1,2,5] , [5], [7,9])

→ [1,2,5,5,7,9]

Dynamic Programming Algorithms

- These are algorithms that “remember” their past results and use them to find new results
- The solution to a problem is obtained by the solutions to sub-problems
- Sub-problems may overlap, and solutions to sub-problems are stored and reused as needed
- This differs from Divide and Conquer, where sub-problems generally need not overlap
- For example, to compute the n^{th} fibonacci number using a Dynamic Programming approach, we store each computed value of $\text{fib}(i)$ for $2 < i \leq n$ in a table T , and we look up $\text{fib}(n-1)$ and $\text{fib}(n-2)$ within T before computing them:

Computing the n^{th} Fibonacci number – dynamic programming

Algorithm `fib`

Input: a non-negative integer n , an initially empty table T

Output: positive integer

```
if (n == 1) or (n == 2)
then { return 1 }
else {
    if value for n-1 is present in T
    then res1 = value for n-1 in T
    else res1 = fib(n-1);
    if value for n-2 is present in T
    then res2 = value for n-2 in T
    else res2 = fib(n-2);
    store (n, res1+res2) in T;
    return res1+res2 }
```

Computing the n^{th} Fibonacci number – dynamic programming

- E.g.

fib(6)

→ fib(5) + fib(4)

→ (fib(4) + fib(3)) + fib(4)

→ ((fib(3) + fib(2)) + fib(3)) + fib(4)

→ (((fib(2) + fib(1)) + fib(2)) + fib(3)) + fib(4)

→ ((2 + fib(2)) + fib(3)) + fib(4) STORE (3,2) in T

→ (3 + fib(3)) + fib(4) STORE (4,3) in T

→ 5 + fib(4) STORE (5,5) in T

→ 8 STORE (6,8) in T

Backtracking Algorithms

- These are based on a depth-first search
- This recursively selects one sub-problem of the overall problem to tackle at a time
- If a overall solution is found, this is returned
- If a solution cannot be found at any point, then the current sub-problem P is abandoned, and an alternative solution is attempted for the previously considered sub-problem P^{prev} – this is called ***backtracking***
- The computation continues from that new solution to P^{prev} , attempting to find a solution again for P
- An example is the ***map colouring problem***, defined as follows
- To start it off, the algorithm `map-colour` is called with `i == 1`:

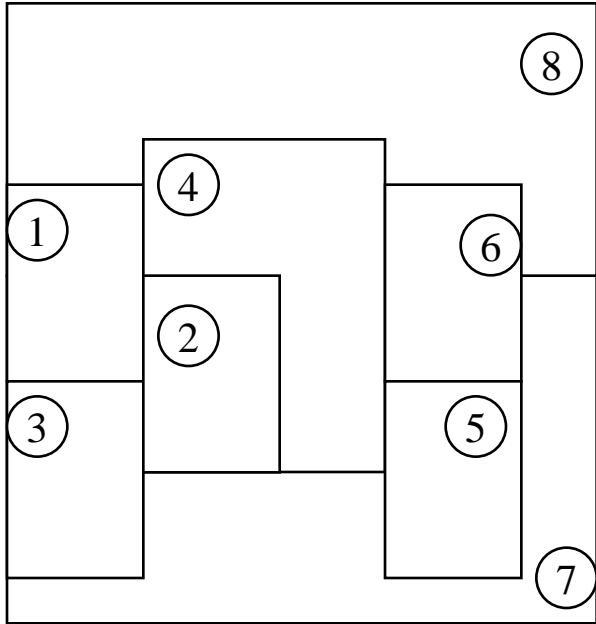
Algorithm **map-colour** (**M**, **i**, **C**)

Input: a map **M** showing **n** countries; a country **i**; and a list of colours

C = [Red, Yellow, Blue, Green]

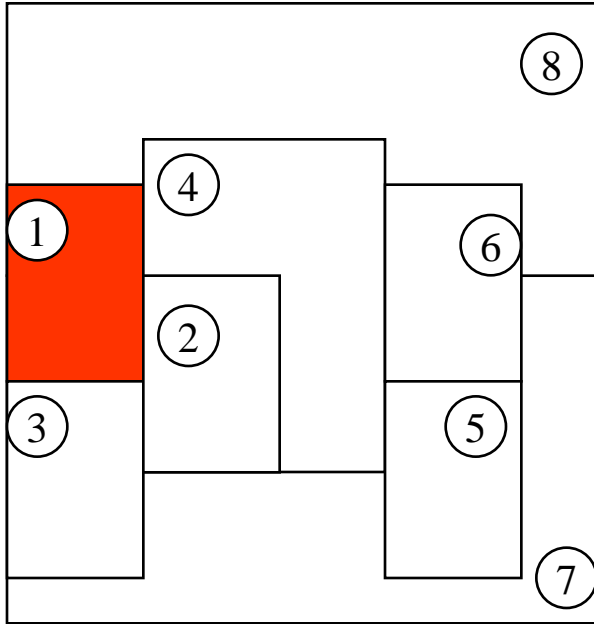
Output: map **M** coloured so that no country is adjacent to a country with the same colour

```
if i == n+1 /* countries finished and M coloured */
then return Success
else {
    for each colour c in C {
        if country i is not adjacent to a country
            that has been coloured c
        then {colour country i with colour c;
            if map-colour (M, i+1, C) == Success
            then return Success};
    }
return Failure
}
```



1	2	3	4	5	6	7	8

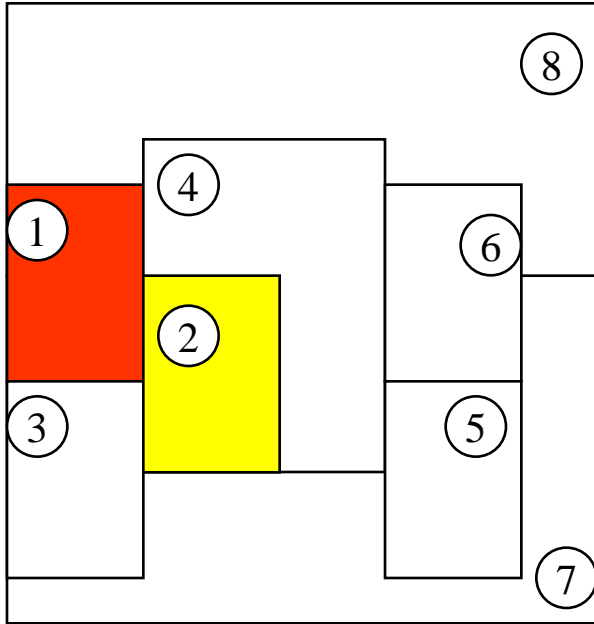
C = [Red, Yellow, Blue, Green]



1	2	3	4	5	6	7	8

C = [Red, Yellow, Blue, Green]

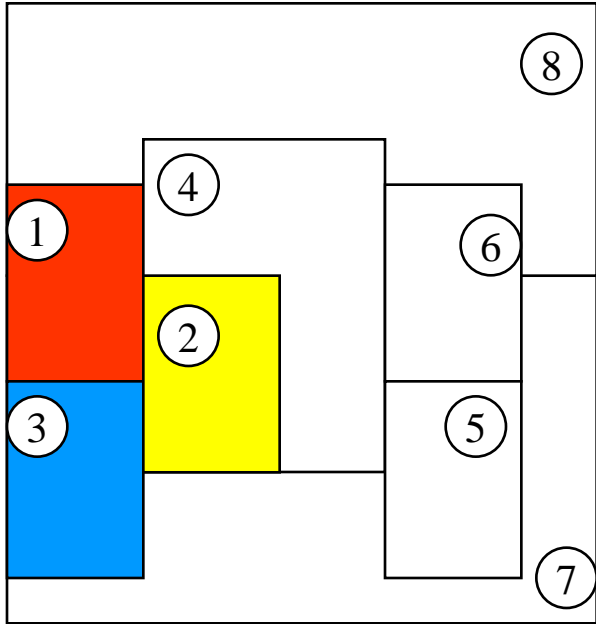
Success for 1



1	2	3	4	5	6	7	8

C = [Red, Yellow, Blue, Green]

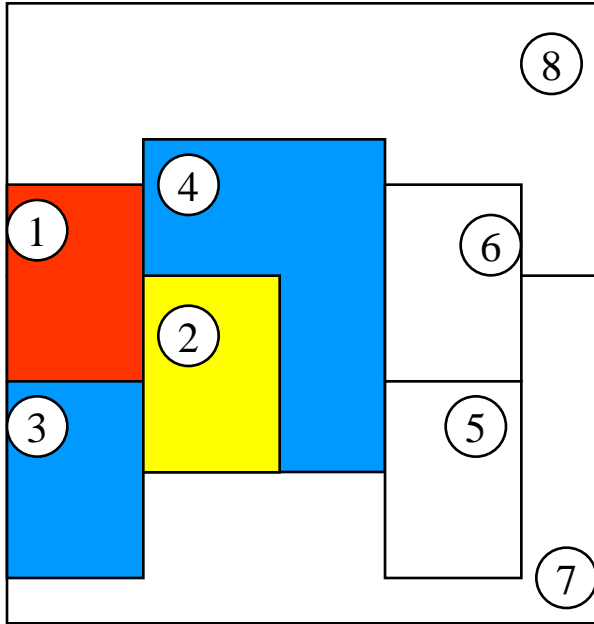
Success for 2



1	2	3	4	5	6	7	8
Red	Yellow	Blue					

C = [Red, Yellow, Blue, Green]

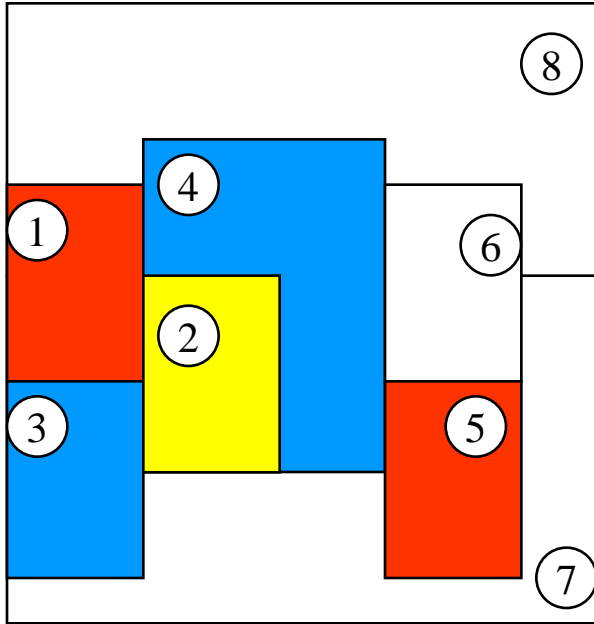
Success for 3



1	2	3	4	5	6	7	8

C = [Red, Yellow, Blue, Green]

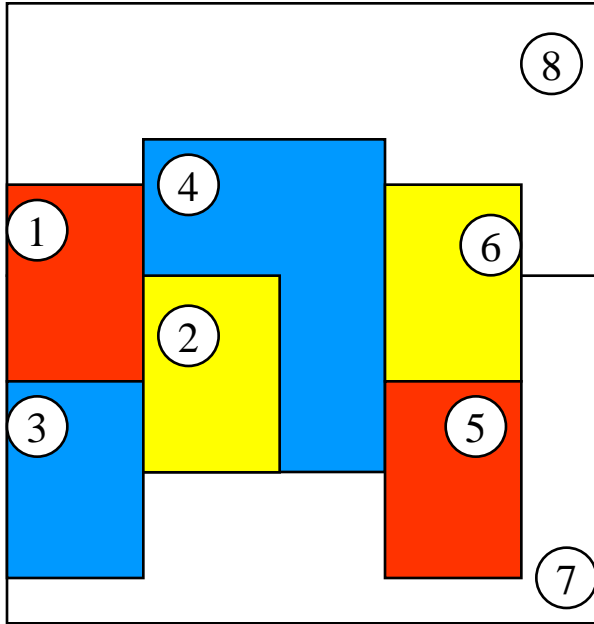
Success for 4



1	2	3	4	5	6	7	8

C = [Red, Yellow, Blue, Green]

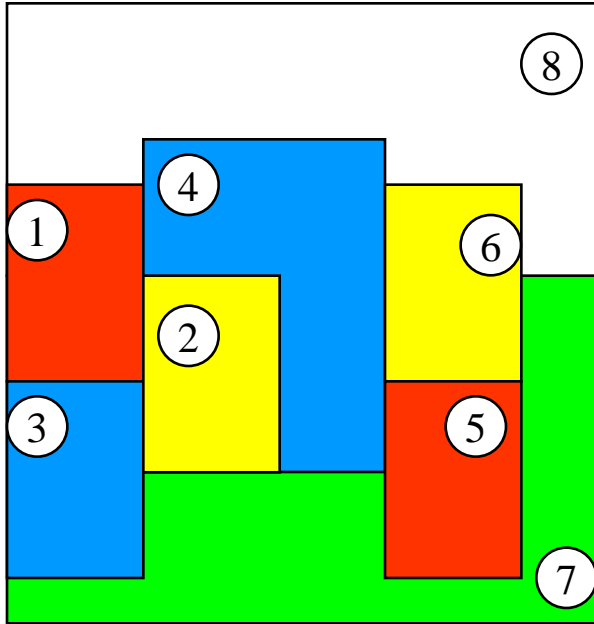
Success for 5



1	2	3	4	5	6	7	8
Red	Yellow	Blue	Blue	Red	Yellow		

C = [Red, Yellow, Blue, Green]

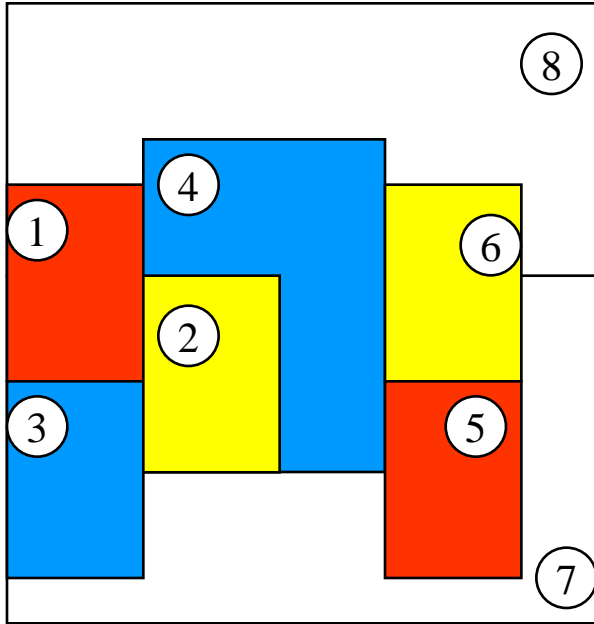
Success for 6



1	2	3	4	5	6	7	8
Red	Yellow	Blue	Blue	Red	Yellow	Green	

C = [Red, Yellow, Blue, Green]

Success for 7



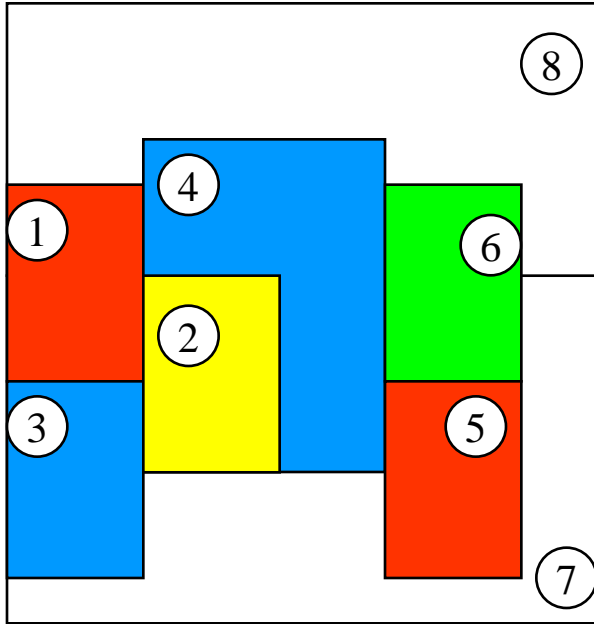
1	2	3	4	5	6	7	8
Red	Yellow	Blue	Blue	Red	Yellow		

$C = [\text{Red, Yellow, Blue, Green}]$

Failure for 8.

Failure for 7.

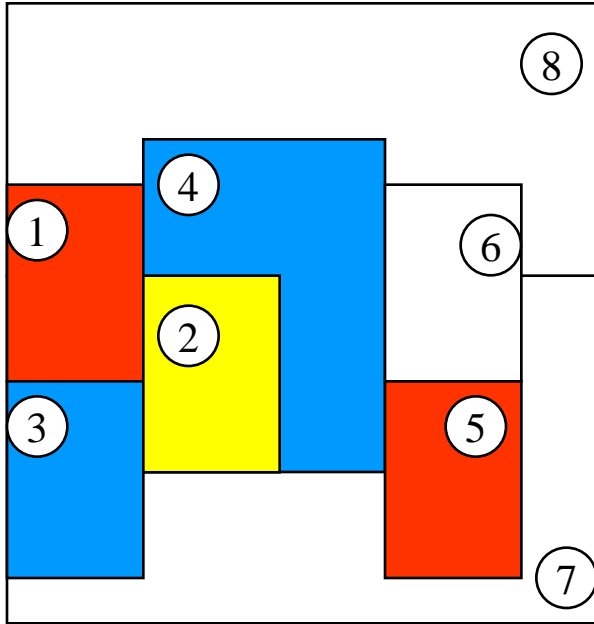
Backtrack to 6.



1	2	3	4	5	6	7	8
Red	Yellow	Blue	Blue	Red	Green		

C = [Red, Yellow, Blue, Green]

Success for 6



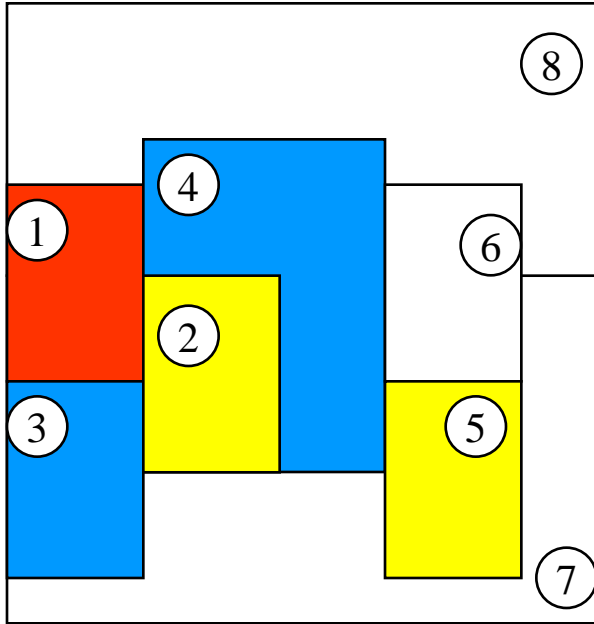
1	2	3	4	5	6	7	8

$C = [\text{Red, Yellow, Blue, Green}]$

Failure for 7.

Failure for 6.

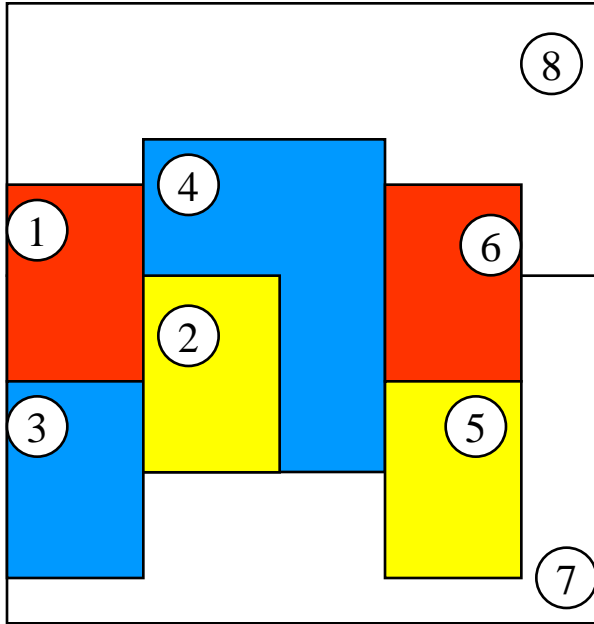
Backtrack to 5.



1	2	3	4	5	6	7	8

C = [Red, Yellow, Blue, Green]

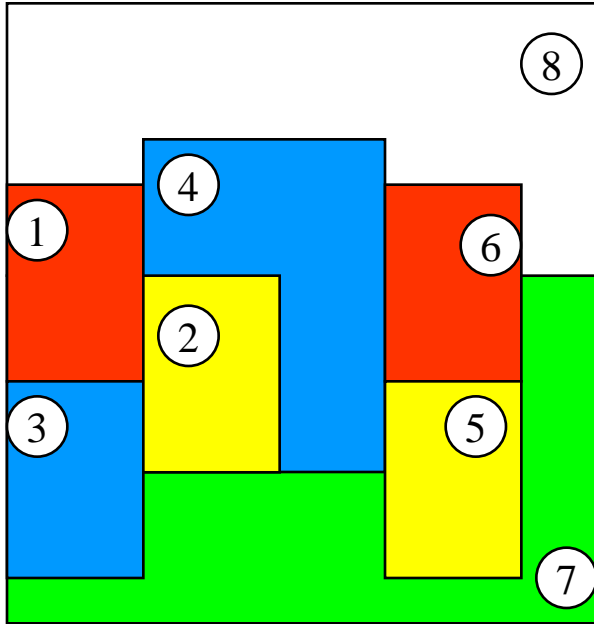
Success for 5



1	2	3	4	5	6	7	8
Red	Yellow	Blue	Blue	Yellow	Red		

C = [Red, Yellow, Blue, Green]

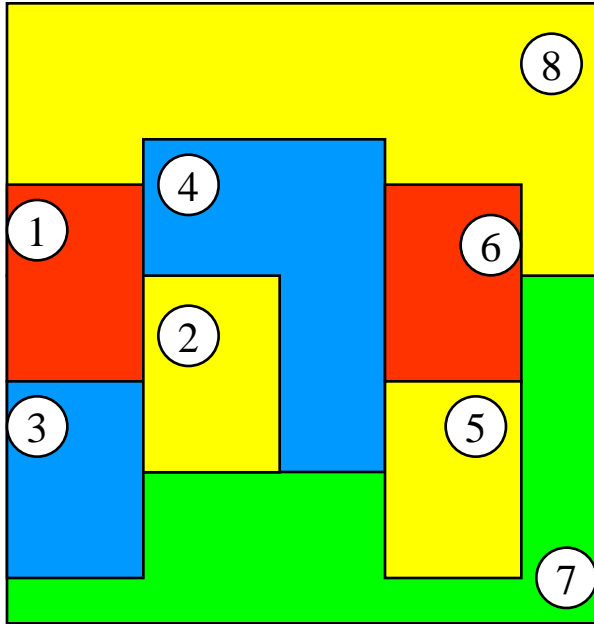
Success for 6



1	2	3	4	5	6	7	8
Red	Yellow	Blue	Blue	Yellow	Red	Green	

C = [Red, Yellow, Blue, Green]

Success for 7



1	2	3	4	5	6	7	8
Red	Yellow	Blue	Blue	Yellow	Red	Green	Yellow

C = [Red, Yellow, Blue, Green]

Success for 8

Finish!

(vi) Implementation and unit testing phase

- ***Implementation*** means translating the detailed design for each program unit into code that is written in a specific programming language
- The objective is to produce code that complies with the detailed design, while at the same time aiming for desirable features of code such as:
 - correctness i.e. meeting the specification
 - reliability
 - produced on time and within the budget
 - readability and maintainability
 - reusability and extensibility
 - traceability (i.e. code is traceable back to the design)
 - efficiency

Implementation and unit testing phase

- Testing of each program unit occurs as the unit is being implemented (testing will be discussed later in this ISD module)
- Testing aims to
 - ***verify*** that the program unit satisfies its specification, and to
 - ***validate*** that the program unit satisfies the functional and non-functional requirements relating to that unit
- Overall, this activity is known as ***V&V*** (verification and validation)
- Testing cannot show the absence of errors, only their presence. Thus, the main aim of testing is to DISCOVER ERRORS in a product, not to confirm that it has no errors!

Implementation and unit testing phase

- Testing never ends but continues throughout the lifetime of a product
- However, the earlier errors are found, the more cheaply and quickly can they be fixed
- Note that designing and implementing are ***constructive*** activities while testing is a ***destructive*** activity
- Different approaches are needed for these two kinds of activities:
 - in the first case, one tries to produce a quality product
 - in the second case one tries to locate its flaws
- This switch from a constructive to a destructive mind-set can be quite hard to achieve!

(vii) System testing

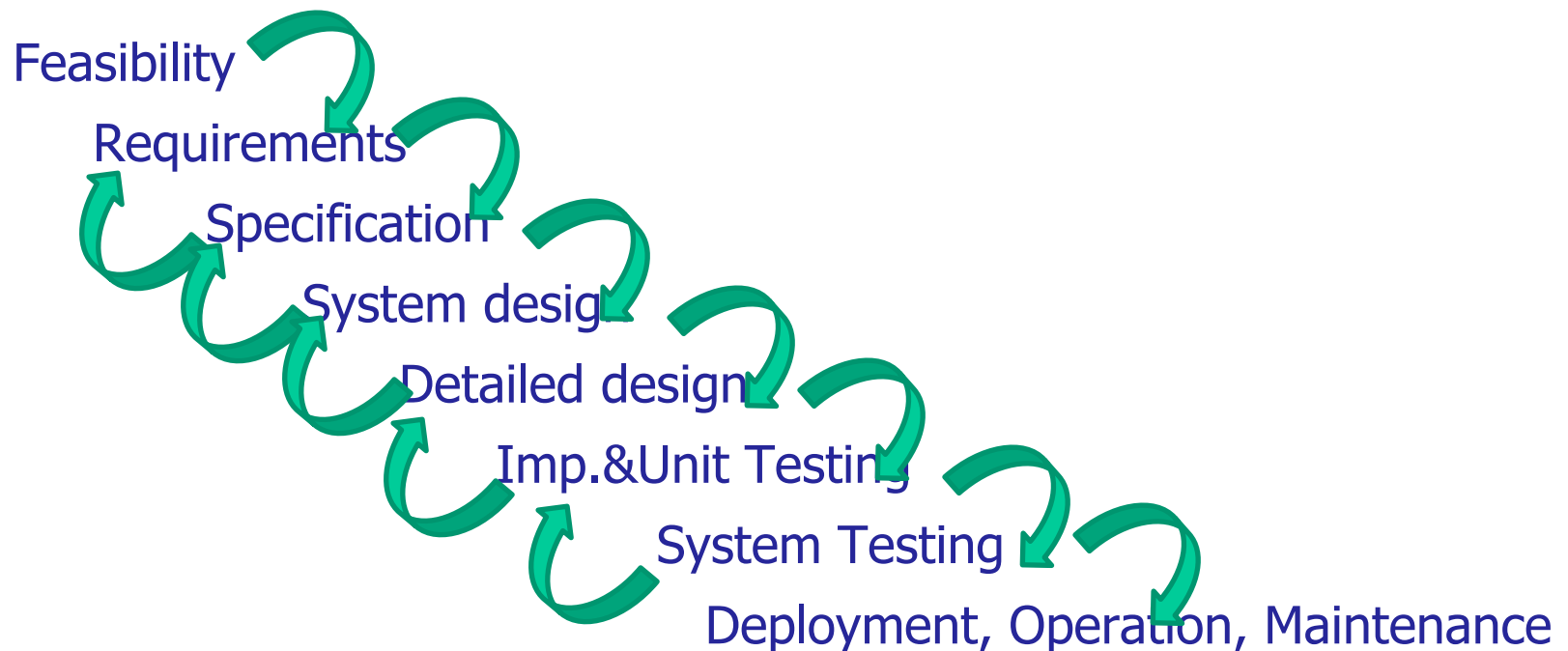
- This is testing undertaken for the system as a whole
- It may occur at the development site – this is known as Alpha testing
- In large-scale commercial software development, an independent test team typically takes over at the System Testing stage, in order to ensure an independent assessment of the system's quality
- The test team is typically under different management ("Quality Assurance") than the development team is ("Software Development")
- A further phase of system testing may occur at customer sites – this is known as Beta testing
- System testing assesses both functional requirements ("what the system does") and non-functional requirements ("how well it does it")

(viii) Deployment, Operation, Maintenance

- A parallel run of a new software system within the existing organisational procedures is often undertaken in order to ensure that the new system conforms to expectations – this is termed Acceptance testing
- The system is then deployed
- From time to time it may need to be corrected or updated – this is termed software maintenance, and typically takes up the bulk of the time spent on software development

2. Software Development Methodologies

- The ordering and repetition of the different software development phases depends on the ***software development methodology*** that is being followed to develop a given software system e.g.
 - the ***Waterfall methodology*** undertakes each phase in turn. There are feedback loops between the phases, due to the inherently iterative nature of software development:



Software Development Methodologies

- ***Incremental development*** divides the system into a number of parts and develops each part separately e.g. by applying the Waterfall methodology to each increment
- ***Prototyping*** can be used to clarify the specification and/or the design of a new system. Development focusses on the parts of the system where there is a lack of clarity of what is required, or how it should be implemented. Simplifications are made for other parts of the system that not the focus of the prototyping activity.
 - The software produced by prototyping can be discarded – this is called “throw-away” prototyping
 - Or, if it is of sufficient quality, the prototype software can be further evolved into part or all of the final system to be delivered
 - Prototyping can be used as a sub-process within any of the other methodologies

Software Development Methodologies

- ***Agile development*** is an incremental development approach:
 - all phases of the lifecycle are the collective responsibility of an integrated development team that involves users, business and systems analysts, developers and QA
 - each increment is quite short (typically a few weeks) and it involves the whole lifecycle, including acceptance testing by the users
 - the overall project is thus quite responsive to changes – hence the term “agile” – because it is delivered in small increments and users are core members of the development team
 - eXtreme programming is an example of an Agile method, and is discussed later in this ISD module

Summary

This lecture has covered:

- The Software Development Process
 - phases involved
 - features of “good” software design
 - algorithm design, pseudocode, types of algorithms

- Software Development Methodologies
 - waterfall, incremental, prototyping, agile

- Deriving Designs – ran out of time, to be covered at the beginning next time

Background Reading

- More information on the Systems Development Lifecycle can be found in Dave Wilson's Information Systems module notes
- The major phases of the lifecycle are discussed, as are development methodologies
- Also discussed are the project team roles on Systems Development projects
- OODP module notes by Eli Katsiri
- Roger S. Pressman. Software Engineering, A practitioner's approach. 6th Edition, 2005