L22 - testing strategy

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

UNIVERSITY OF ABERDEEN

22.1 Reminding past issues and mapping them to current topics

Where are we now?

⇒ · · · · · ·

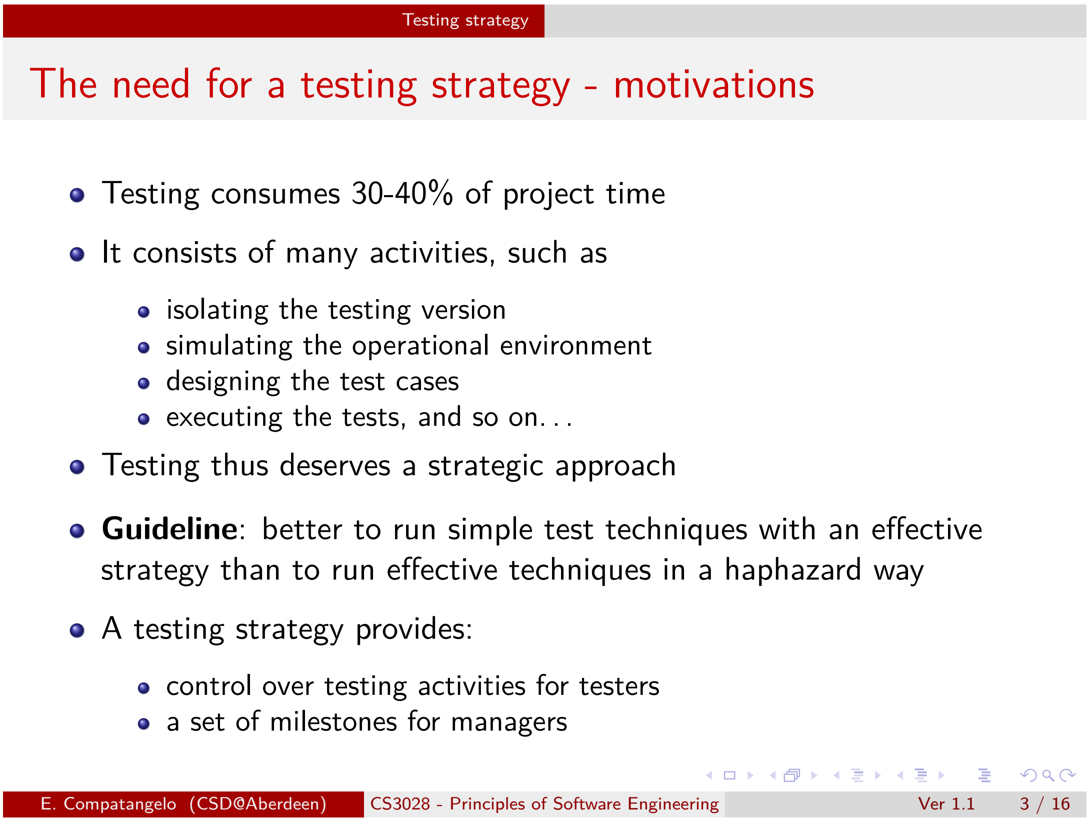
⇒ Elaboration

⇒ Construction (third UP phase)

- ⇒ Software Quality Control
  - ⇒ Software Testing
    - ⇒ · · · · · ·
    - ⇒ Testing Strategy

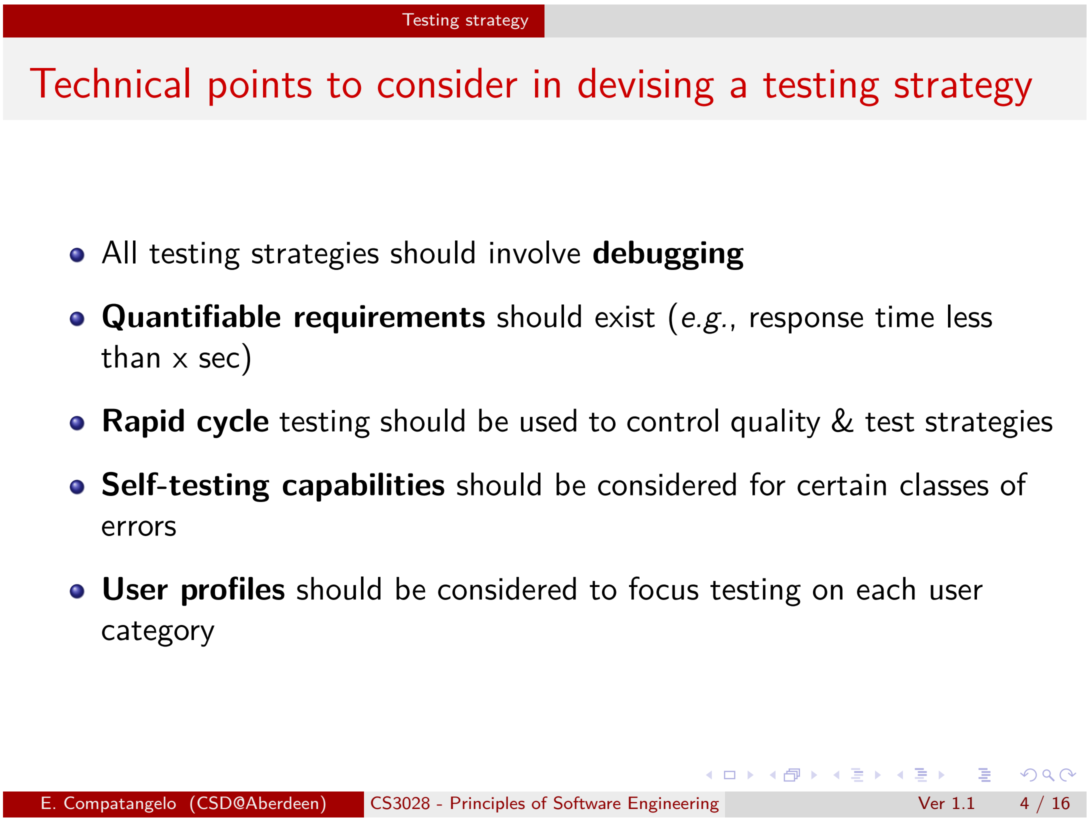
⇒ · · · · · ·

22.2 Motivations, objectives, issues



The need for a testing strategy - motivations

- Testing consumes 30-40% of project time
- It consists of many activities, such as
  - isolating the testing version
  - simulating the operational environment
  - designing the test cases
  - executing the tests, and so on. . .
- Testing thus deserves a strategic approach
- **Guideline**: better to run simple test techniques with an effective strategy than to run effective techniques in a haphazard way
- A testing strategy provides:
  - control over testing activities for testers
  - a set of milestones for managers



Technical points to consider in devising a testing strategy

- All testing strategies should involve **debugging**
- **Quantifiable requirements** should exist (*e.g.*, response time less than x sec)
- **Rapid cycle** testing should be used to control quality & test strategies
- **Self-testing capabilities** should be considered for certain classes of errors
- **User profiles** should be considered to focus testing on each user category

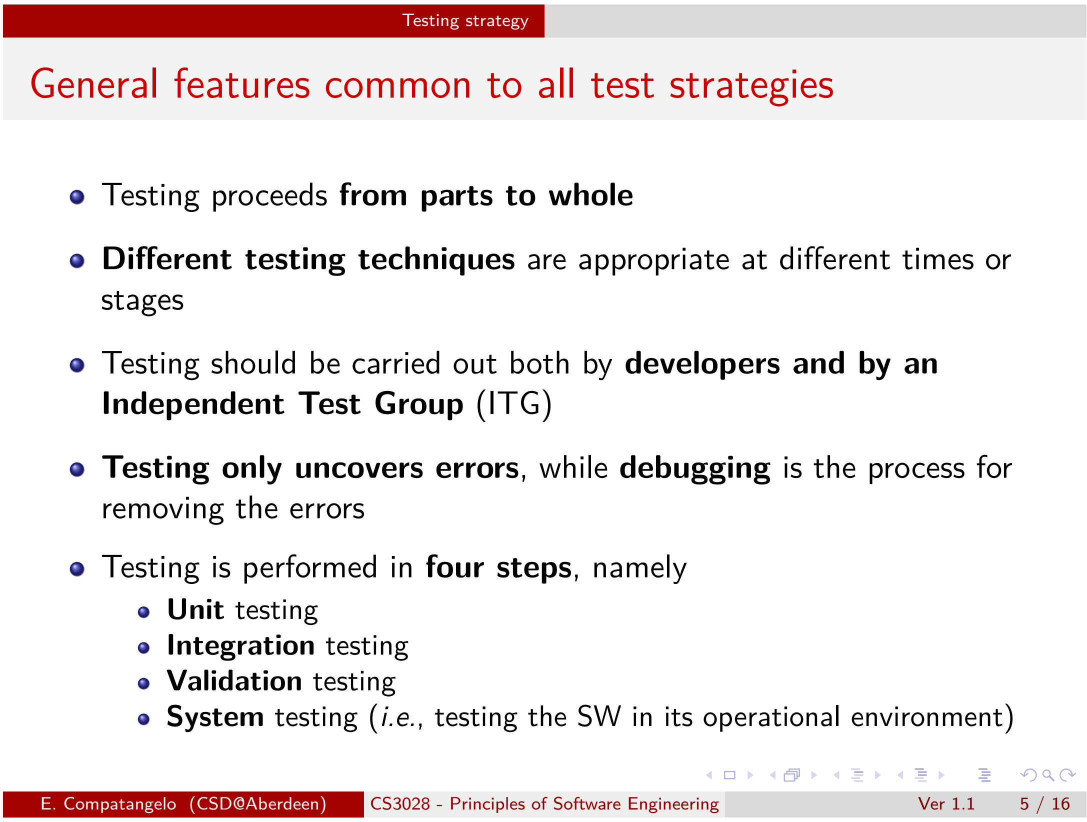
22.2.1 Managerial points to consider in devising a testing strategy

- A testing strategy provides a **template of test steps** (unit, integration, validation etc) to be thoroughly managed
- Appropriate testing techniques (black or white box) should be used in each test step
- Testing objectives should be stated upfront in **measurable terms**
- Test strategy and test cases should undergo **formal technical review**
- Formal technical reviews should be used to **reduce the effort needed for testing**
- Continuous improvement of the testing process should be helped by using **metrics**

22.3 Strategies for testing strategy

22.3.1 Overall test strategy

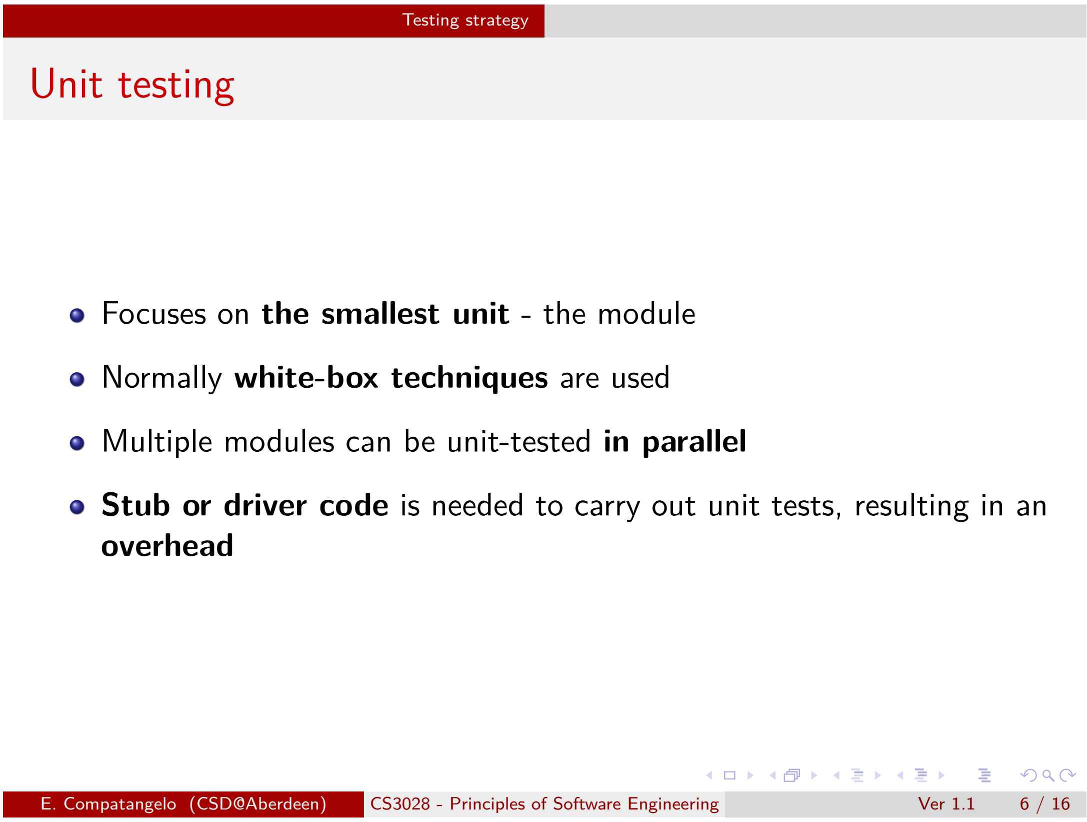
- Set up and use a **template** for testing
- **Many strategies** are possible
- Choice of an **effective strategy** depends upon
  - Testing **objectives**
  - Other **QA tasks** such as technical reviews
  - **Development process**
  - Nature of the **project**



General features common to all test strategies

- Testing proceeds **from parts to whole**
- **Different testing techniques** are appropriate at different times or stages
- Testing should be carried out both by **developers and by an Independent Test Group** (ITG)
- **Testing only uncovers errors**, while **debugging** is the process for removing the errors
- Testing is performed in **four steps**, namely
  - **Unit** testing
  - **Integration** testing
  - **Validation** testing
  - **System** testing (*i.e.*, testing the SW in its operational environment)

22.4 Testing steps: unit, integration, validation, and system testing

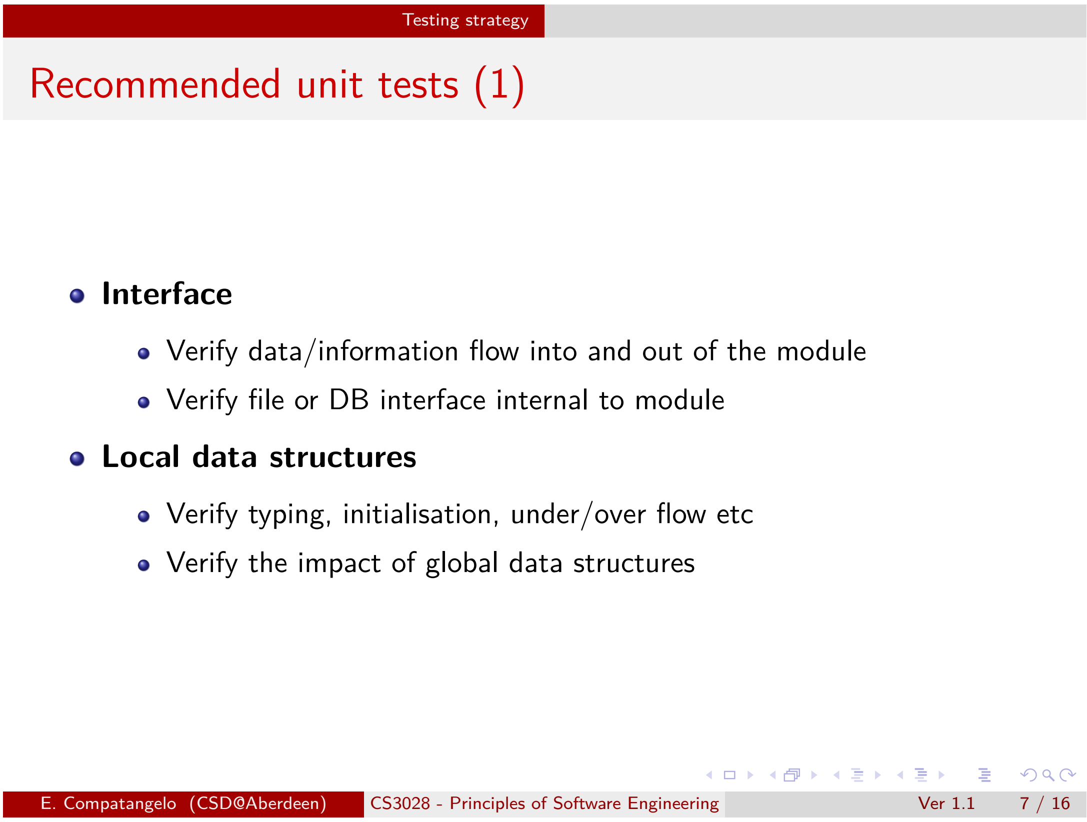


Testing strategy

Unit testing

- Focuses on **the smallest unit** - the module
- Normally **white-box techniques** are used
- Multiple modules can be unit-tested **in parallel**
- **Stub or driver code** is needed to carry out unit tests, resulting in an **overhead**

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 6 / 16



Testing strategy

Recommended unit tests (1)

- **Interface**
  - Verify data/information flow into and out of the module
  - Verify file or DB interface internal to module
- **Local data structures**
  - Verify typing, initialisation, under/over flow etc
  - Verify the impact of global data structures

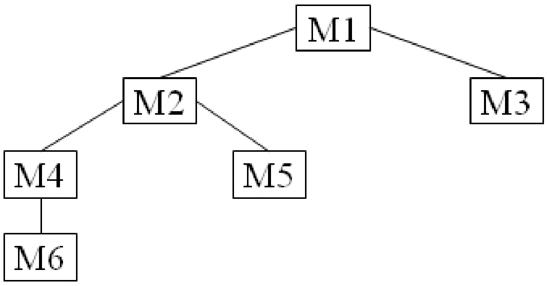
E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 7 / 16

Recommended unit tests (2)

- **Boundary conditions**
  - Verify *just below*, *at*, and *just above* minima and maxima
  - for the *n*-th element of an *n*-dimensional array
  - for the *i*-th iteration of a loop with *i* passes
  - For *Max* and *min* data values
- **Independent paths**
  - Basis paths
  - Loop tests
  - Condition tests
- **Error handlers**

Integration testing

- Once integrated, independently unit-tested modules need additional testing
- Big-bang integration should be avoided in favour of incremental integration, which is (apparently) more time- and thus resource-expensive
- Integration testing is generally performed in a top-down fashion:
  - Depth first
  - Breadth first

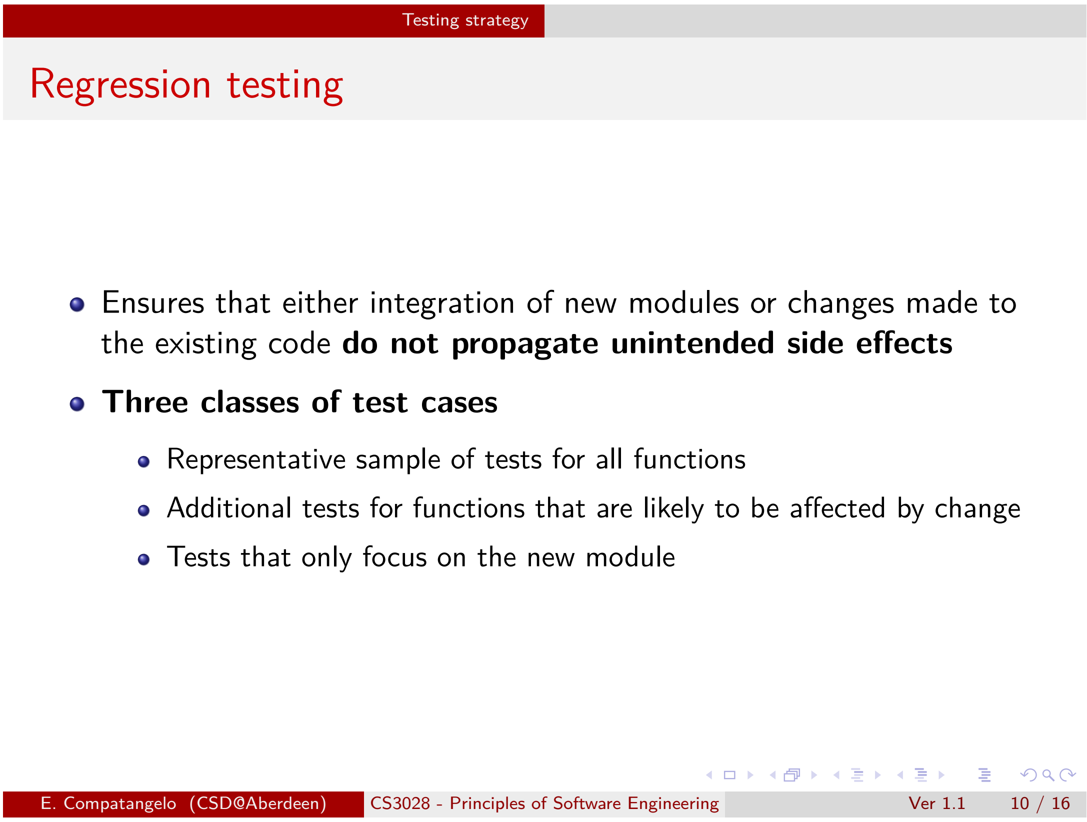


22.4.1 Top-down integration

- Initially **stubs replace lower-level modules**
- Tests are carried out **as if all the modules were integrated**
- Later, lower-level module stubs are replaced by the actual modules on a level-by-level basis
- Stubs lead to significant **overheads**

22.4.2 Bottom-up integration

- Low-level modules are combined into **clusters**
- **Drivers** coordinate tests
- Integration process moves up **combining larger clusters** for every integration step
- A number of drivers is needed; this can be reduced by combining top-down with bottom-up integration processes (*sandwich*)

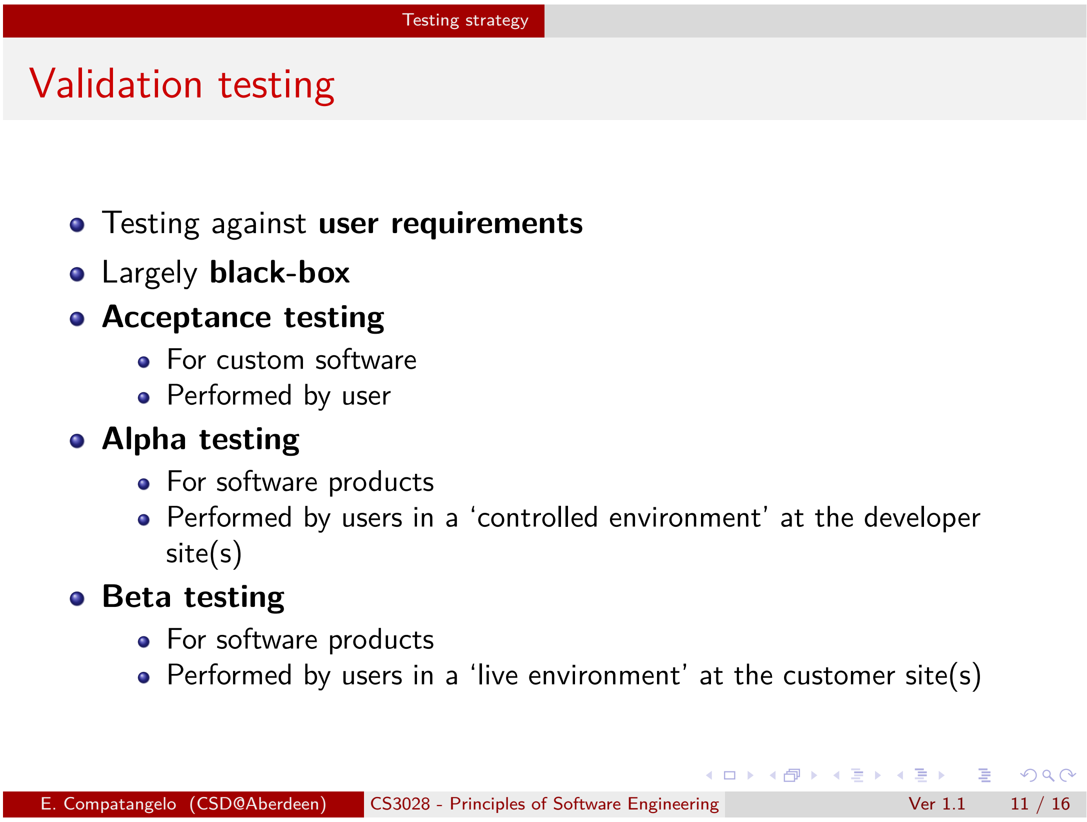


Testing strategy

Regression testing

- Ensures that either integration of new modules or changes made to the existing code **do not propagate unintended side effects**
- **Three classes of test cases**
  - Representative sample of tests for all functions
  - Additional tests for functions that are likely to be affected by change
  - Tests that only focus on the new module

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 10 / 16

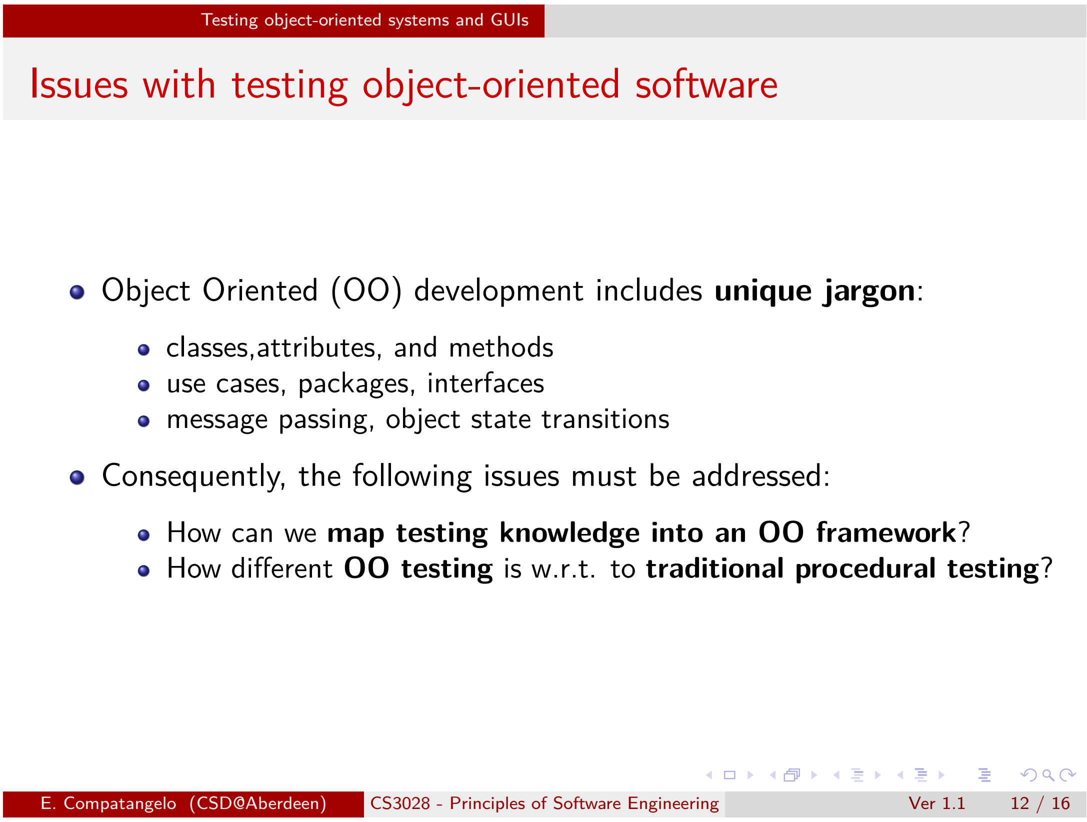


Validation testing

- Testing against **user requirements**
- Largely **black-box**
- **Acceptance testing**
  - For custom software
  - Performed by user
- **Alpha testing**
  - For software products
  - Performed by users in a ‘controlled environment’ at the developer site(s)
- **Beta testing**
  - For software products
  - Performed by users in a ‘live environment’ at the customer site(s)

23 Testing object-oriented systems and GUIs

23.1 The peculiarities of object-oriented software and of its testing



Testing object-oriented systems and GUIs

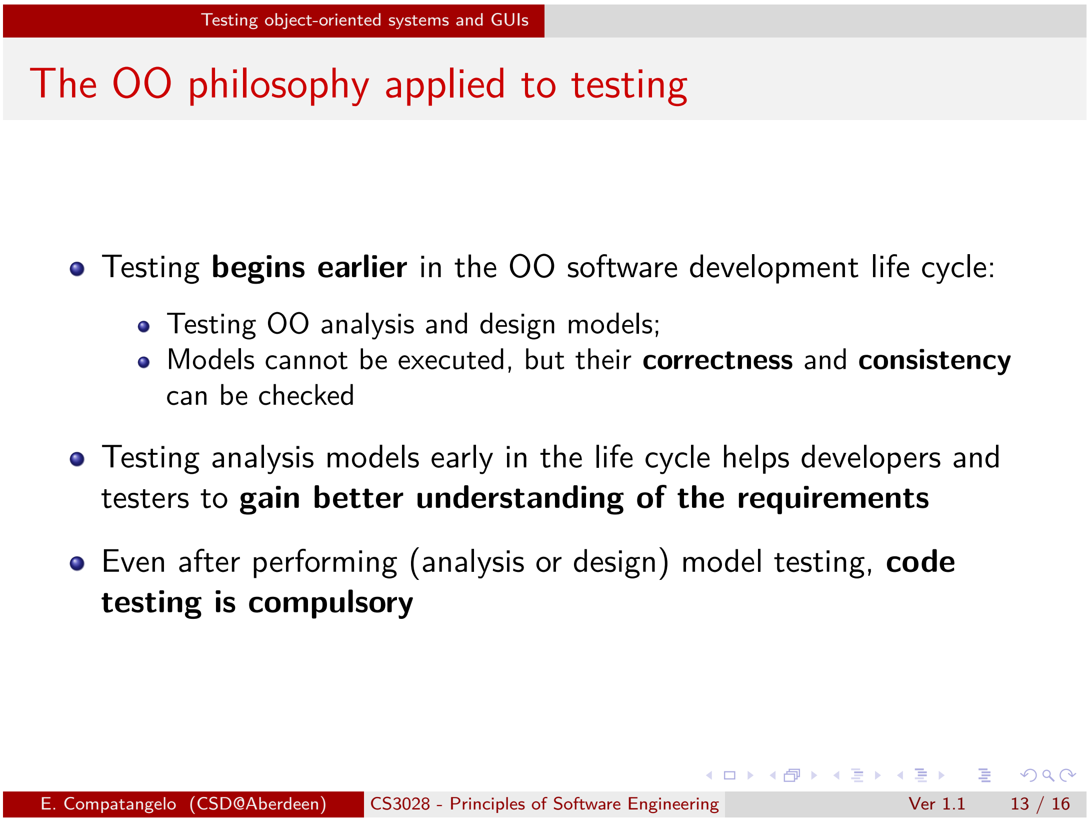
Issues with testing object-oriented software

- Object Oriented (OO) development includes **unique jargon**:
  - classes,attributes, and methods
  - use cases, packages, interfaces
  - message passing, object state transitions
- Consequently, the following issues must be addressed:
  - How can we **map testing knowledge into an OO framework**?
  - How different **OO testing** is w.r.t. to **traditional procedural testing**?

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 12 / 16

23.1.1 Object-oriented software and testing

- OO **analysis, design and code levels** involve **similar semantic constructs** (*i.e.* classes, attributes, operations and messages)
- **Consequently**, there is a **low semantic gap** between analysis, design and code models:
  - OO analysis models can be easily transformed into OO design models
  - OO design models in turn can be transformed into OO code
- **Implications** for OO testing:
  - **Develop** tests for OO analysis models
  - **Refine** them for OO design models
  - **Refine them further** for OO code



Testing object-oriented systems and GUIs

The OO philosophy applied to testing

- Testing **begins earlier** in the OO software development life cycle:
  - Testing OO analysis and design models;
  - Models cannot be executed, but their **correctness** and **consistency** can be checked
- Testing analysis models early in the life cycle helps developers and testers to **gain better understanding of the requirements**
- Even after performing (analysis or design) model testing, **code testing is compulsory**

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 13 / 16

23.2 Object-oriented software testing strategy

23.2.1 Object-oriented test strategy

- The **classic strategy** is still valid:
  - unit testing
  - integration testing
  - validation testing
  - system testing
- **Individual steps in the strategy may need some changes and/or adaptations**
- OO testing ideas are still evolving
- However, most testing concepts are applicable to OO software with minor or no modifications at all

23.2.2 Unit testing in object-oriented software

- **unit = class**
- Classes **encapsulate data and operations**, hence:
  - Data can no longer be seen as flowing across the unit interface
  - A unit is not just the procedure (algorithm) but also includes all class elements, including object states

- Classes are organised into **hierarchies**, hence
  - Testing methods in the abstract class is pointless
  - Methods are invoked (and should be thus tested) in the private context of subclasses

23.2.3 Integration testing in object-oriented software

- **Conventional** top-down and bottom-up integration strategies hard to implement, because of
  - **Lack of strict hierarchical control**
  - **Multiple threads**
  - **Interdependencies among class components** (attributes and methods) that make it hard to integrate one of them at a time – *interclass* high coupling & cohesion
- New integration testing strategies are needed
- **Thread-based** testing:
  - **Test each thread** (classes that respond to one input)
  - **Apply regression** testing to check side-effects
- **Use-based** testing
  - **Test independent classes first**
  - **Then test dependent classes**

23.3 Test case design for object-oriented software

23.3.1 Test case design for OO software (1)

- **Black-box testing techniques valid for any software**:
  - not based on code, but on input domain and requirements
  - use cases can drive black-box testing
- **White-box techniques valid for methods in a class**
- **Newer methods** are needed:
  - Newer method: **fault-based testing** — predict the faults based on the analysis model
  - Newer method: **scenario based testing** — variations of use cases

23.3.2 Test case design for OO software (2)

- Newer method: **random testing**
  - Operations defined in a class may have to be called under sequential constraints (*e.g.*, in a banking application, an account must be opened before applying other operations)
  - Even with such constraints many possible sequences of calls are possible
- Newer method: **partition testing**
  - Reduces the No of test cases by partitioning some aspect of the object
  - Partition state operations and non-state operations in different sets
  - Partition attribute values into equivalence classes (see earlier)

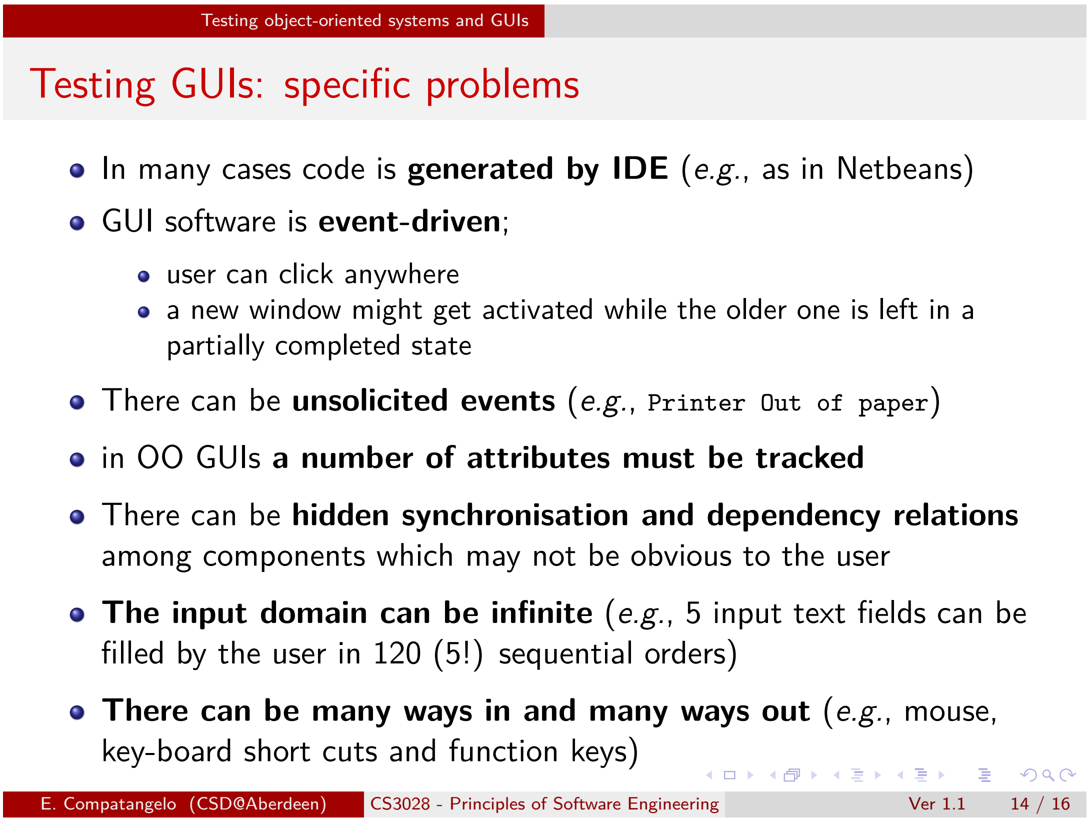
23.3.3 Test case design for OO software (3)

- Newer method: **state-based testing**
  - based on state machine diagrams
  - for each transition in the state machine diagram, a test case is designed, ensuring that the object is tested in all its states
  - However, since the state of an object is encapsulated test cases should include inputs to move the object into the desired state before executing a test
  - Needs support from tools to make the desired state transitions

23.3.4 Test case design for OO software (4)

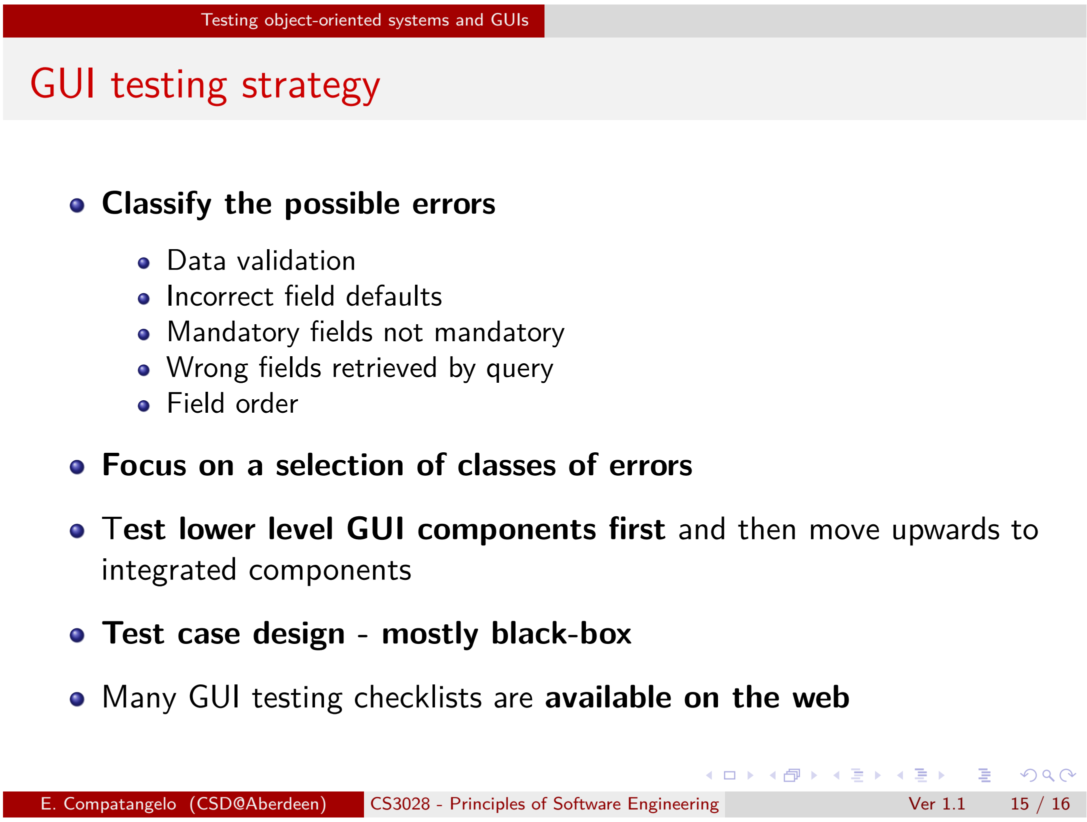
- Newer method: **testing reused classes**
  - Thoroughly tested classes from earlier projects are often reused in OO software
  - Testing the reused classes in the new context is necessary
- Newer method: **polymorphism testing**
  - Polymorphism in OO software is another feature which allows code to be used with several data types or object types
  - Polymorphic methods need to be tested with all the possible object type bindings
- Testing effort in OO software is increased by class reuse and polymorphism

23.4 GUI testing



Testing GUIs: specific problems

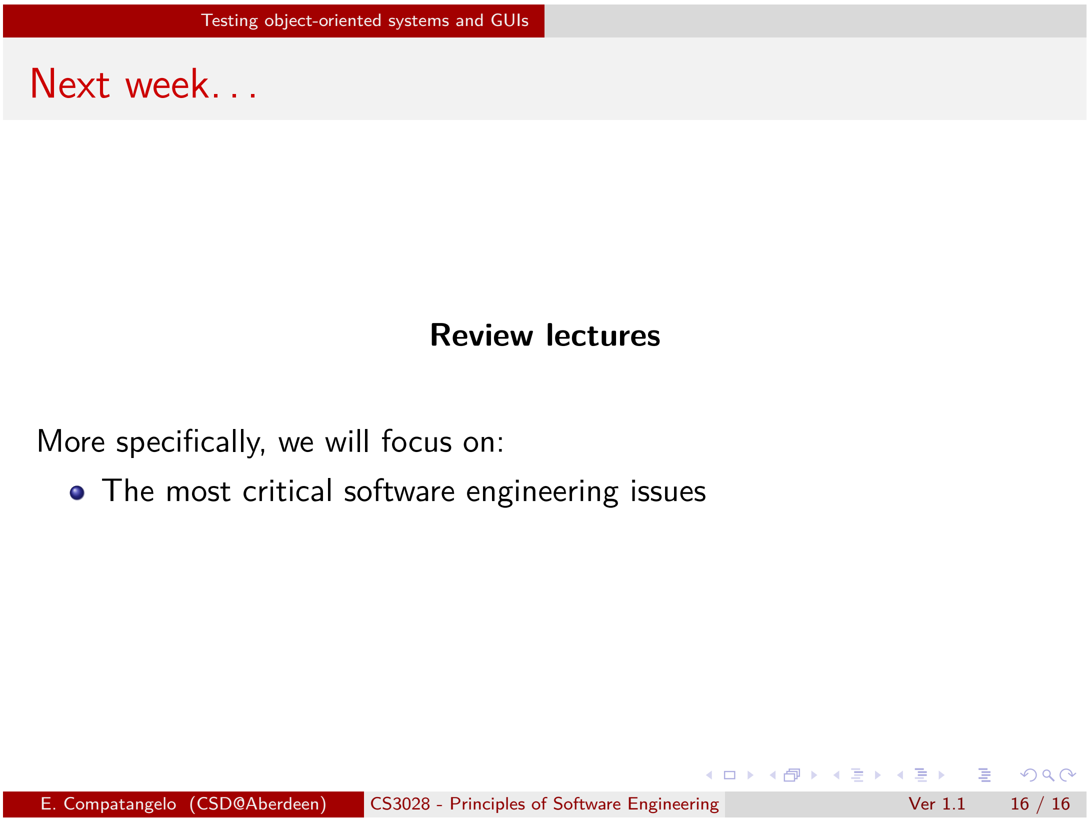
- In many cases code is **generated by IDE** (*e.g.*, as in Netbeans)
- GUI software is **event-driven**;
  - user can click anywhere
  - a new window might get activated while the older one is left in a partially completed state
- There can be **unsolicited events** (*e.g.*, `Printer Out of paper`)
- in OO GUIs **a number of attributes must be tracked**
- There can be **hidden synchronisation and dependency relations** among components which may not be obvious to the user
- **The input domain can be infinite** (*e.g.*, 5 input text fields can be filled by the user in 120 (5!) sequential orders)
- **There can be many ways in and many ways out** (*e.g.*, mouse, key-board short cuts and function keys)



GUI testing strategy

- **Classify the possible errors**
  - Data validation
  - Incorrect field defaults
  - Mandatory fields not mandatory
  - Wrong fields retrieved by query
  - Field order
- **Focus on a selection of classes of errors**
- T**est lower level GUI components first** and then move upwards to integrated components
- **Test case design - mostly black-box**
- Many GUI testing checklists are **available on the web**

23.5 Preparing for the topic ahead



Next week. . .

Review lectures

More specifically, we will focus on:

- The most critical software engineering issues