Well-behaved objects
Main concepts to be covered

- Testing
- Debugging
- Test automation
- Writing for maintainability
We have to deal with errors

• Early errors are usually *syntax errors*.
  – Syntax errors are grammar faults.
  – The compiler will spot these.
• Later errors are usually *logic errors*.
  – The compiler cannot help with these.
  – Also known as bugs (not features 😊).
• Some logical errors have no immediately obvious manifestation.
  – Commercial software is rarely free of errors.
  – Formal proofs of correctness or code verifications are a tough job – but important!
Prevention vs. Detection
(Developer vs. Maintainer)

• We can lessen the likelihood of errors.
  – Use software engineering techniques, such as encapsulation ➔ avoid errors!

• We can improve the chances of detection.
  – Use software engineering practices, such as modularization and documentation ➔ detect errors!

• We can develop detection skills.

• This means experiments – more and more important for informatics.
Testing and debugging

• These are crucial skills.
• **Testing** searches for the presence of errors (in your own as well as in others’ programs).
• **Debugging** searches for the source and location of errors.
  – The manifestation of an error may well occur some ‘distance’ from its source.
  – Another aspect of concern: side effects.
Testing and debugging techniques

- Unit testing (within BlueJ)
- Test automation
- Manual walkthroughs
- Print statements
- Debuggers
Unit testing

• Each unit of an application may be tested individually (in contrast to application testing).
  – Units in this context: methods, classes, modules (packages in Java).

• Unit testing can (should) be done early during development.
  – Finding and fixing errors early lowers development costs (e.g. programmer time).
  – A (relevant/reallistic) test suite is built up.
Testing fundamentals

- Understand what the unit should do – its contract.
  - You will be looking for violations.
  - Use positive tests (what should work well) and negative tests (what should go wrong).
- Test and check boundaries or extreme cases.
  - Zero, One, Full.
    - Search an empty collection.
    - Add to a full collection.
    - Assign zero.
Test automation

- Good testing is a creative process, but ...
- ... thorough testing is time consuming and repetitive.
- **Regression testing** involves re-running tests.
  - Corrections may introduce new errors.
  - Hence: re-start the tests done so far.
- Use of a *test rig* or *test harness* can relieve some of the burden.
  - Special classes are written just to perform the testing.
  - Creativity focused in creating these.
Test automation

• Explore through the *diary-testing* project.
  – Human analysis of the results still required (check the printed results of the test rig).

• Explore fuller automation through the *diary-test-junit* projects.
  – Intervention only required if a failure is reported.
Unit testing within BlueJ

- Objects of individual classes can be created.
- Individual methods can be invoked.
- Inspectors provide an up-to-date view of an object’s state.
- (Explore through the *diary-prototype* project).
Unit testing within Eclipse

- Objects of individual classes can be created.
- Individual methods can be invoked.
- Inspectors provide an up-to-date view of an object’s state.
- Try it in/with the exercises!
Modularization and interfaces

• Applications often consist of different modules.
  – E.g. so that different teams can work on them.
• The *interface* between modules must be clearly specified.
  – Supports independent concurrent development.
  – Increases the likelihood of successful integration later.
Modularization in a calculator

- Each module does not need to know implementation details of the other.
  - User controls could be a GUI or a hardware device.
  - Logic could be hardware or software.
Method signatures as an interface

// Return the value to be displayed.
public int getDisplayValue();

// Call when a digit button is pressed.
public void numberPressed(int number);

// Call when the plus operator is pressed.
public void plus();

// Call when the minus operator is pressed.
public void minus();

// Call to complete a calculation.
public void equals();

// Call to reset the calculator.
public void clear();
Debugging

• It is important to develop code-reading skills.
  – Debugging will often be performed on others’ code.
  – Learning to program means learning to write and to read programs – try some code reading!

• Techniques and tools exist to support the debugging process.
Commenting and style

- Give comments to
  - Classes (purpose, author, version, …)
  - Methods (purpose, return type, parameters, …)
- Blocking structure/layout supports visual understanding:

```
   aaaa aaaa
      bbbb bbbb
         cccc cccc
            dddd dddd
```
- Expressive naming – but not
  
  `TheVariableExpressingTemperature`
Manual walkthroughs

- Relatively underused.
  - A low-tech approach.
  - More powerful than appreciated.
- Get away from the computer!
- ‘Run’ a program by hand.
- High-level (Step) or low-level (Step into) views.
Tabulating object state

• An object’s behaviour is usually determined by its state.
• Incorrect behaviour is often the result of an incorrect state.
• Tabulate the values of all fields.
• Document state changes after each method call.
Verbal walkthroughs

• Explain to someone else what the code is doing.
  – They might spot the error for you.
  – The process of explaining might help you to spot it for yourself.

• Group-based processes exist for conducting formal walkthroughs or inspections.
Print statements

• The most popular technique – even among experts.
• No special tools required.
• All programming languages support them.
• Only effective if the right methods are documented.
• Output may be voluminous!
• Turning off and on requires forethought.
Print statements

• Typical information provided:
  – which methods have been called
  – the values of parameters
  – the order in which methods have been called
  – the values of local variables and fields at strategic points

• Use it to locate an error, then create an automated test for it!
Choosing a test strategy

• Be aware of the available strategies.
• Choose strategies appropriate to the point of development.
• Automate whenever possible.
  – Reduces tedium.
  – Reduces human error.
  – Makes (re)testeing more likely.
Debuggers

• Debuggers are both language- and environment-specific.
  – BlueJ has an integrated debugger (demo).
  – Eclipse has an integrated debugger (demo).
• Support breakpoints.
• Step and Step-into controlled execution.
• Call sequence (stack).
• Object state.
• (Explore through the calculator-engine or debugdemo project).
Review

• Errors are a fact of life in programs.
• Good software engineering techniques can reduce their occurrence.
• Testing and debugging skills are essential.
• Make testing a habit.
• Automate testing where possible.
• Practice a range of debugging skills.