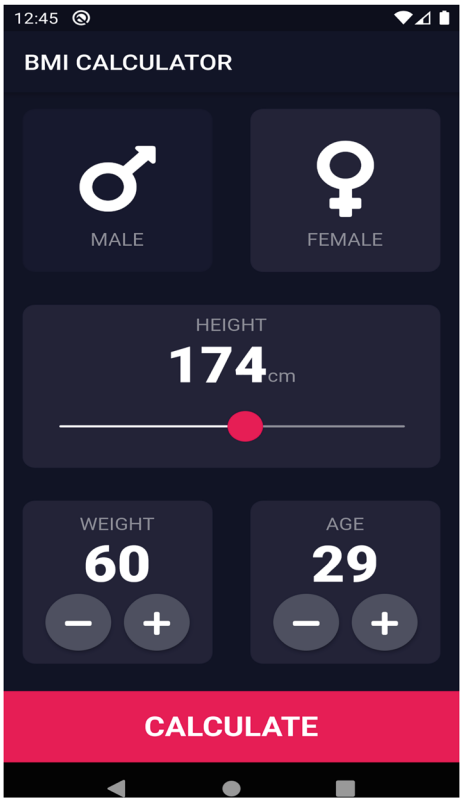

Fig. 4.16 BMI calculator with a different interface (source: https://www.amazon.com/Meet-shingala-Bmi-calculator/dp/B08FJ48KYQ)

In order to try to force a failure from item 1 of the list, the tester can try to enter a very long string of digits into the "weight" input field.[4]

In order to try to force a failure from item 2 of the list, the tester can give zero height, since BMI is calculated as the quotient of weight and the square of height.

In order to try to force a failure from item 3, the tester can give, for example, a negative weight value to force a negative (incorrect) BMI value.

Note that these attacks can be carried out for the above application because its interface allows weight and height values to be entered directly from the keyboard. It is a good idea to design interfaces in a way that prevents incorrect values from being entered. For example, by using fields with built-in value range controls, we allow the user to enter the input only through buttons that allow values to be increased or decreased in a controlled range. Another idea is to use mechanisms such as drop-down lists or sliders. Figure 4.16 shows such an interface for an analogous BMI calculator app.

[4]The formula for BMI divides weight by the square of height. To get an overflow of BMI values, the numerator of the fraction should be as large as possible. Therefore, it does not make sense to give large values for height, because the larger the denominator, the smaller the BMI.

4.4.2 Exploratory Testing

Description of the Technique

Exploratory testing 🕮 is the approach of designing, executing, and recording unscripted tests (i.e., tests that have not been designed in advance) and evaluating them dynamically during execution. This means that the tester does not execute tests prepared in a separate design or analysis phase, but each of their next steps in the current test scenario is dynamic and depends on:

- Knowledge and intuition of the tester
- Experience with this or similar applications
- The way the system behaved in the previous step of the scenario

In fact, the strict division between exploratory and *scripted* testing does not quite capture the truth. This is because every testing activity includes some element of planning and some element of dynamics and exploration. For example, even in pure exploratory testing, it is usual to plan an exploratory session in advance, allocating, for example, a specific time for the tester to execute it. An exploratory tester may also prepare various test data needed for testing before the session starts.

Session-Based Exploratory Testing

Session-based exploratory testing is an approach that allows testers and test managers to obtain greater structuring of the tester's activity, as well as the possibility of better managing *exploratory testing activities*. It consists of three basic steps:

1. The tester meets with the manager, to determine the scope of testing and to allocate time. The manager hands the *test charter (see Table* 4.13*) to the tester or writes it collaboratively with the tester*. The test charter should be created during the test analysis (see Sect. 1.4.3).
2. Conducting an exploratory testing session by the tester, during which the tester makes notes of any relevant observations (problems observed, recommendations for the future, information on what failed during the session, etc.).
3. Meeting again at the end of the session, where the results of the session are discussed and decisions are made on possible next steps.

Session-based exploratory testing takes place in a well-defined time frame (usually from 1 to 4 h). The tester focuses on the task specified in the test charter, but can of course—if necessary—deviate to some extent from the essential task, in a situation where some serious defect in another area of the application is observed. The results of the session are documented by the tester in the test charter.

It is recommended to perform session-based exploratory testing in a collaborative fashion, for example, by using pairing. A tester might pair with a business representative, end user, or Product Owner, to explore the application while testing. A tester might just as well pair with a developer, to avoid testing features that are unstable, to address certain technical issue, or to demonstrate unwanted behavior immediately, without having to provide extensive evidence in the defect report.

Table 4.13 Test charter

Test charter—TE 02-001-01	
Goal	Test the login functionality
Areas	Log in as an existing user with a correct login and password Log in as an existing user through a Google account Log in as an existing user through a Facebook account Incorrect login—no user Incorrect login—wrong password Actions that result with the blocking of the account Using the password reminder function SQL injection attack[a] and other security attacks
Environment	Site accessible through various browsers (Chrome, FF, IE)
Time	2019-06-12, 11:00–13:15
Tester	Bob Bugbuster
Tester's notes	
Files	(Screenshots, test data, etc.).
Defects found	
Division of time	20% preparation for the session 70% conduct of session 10% problem analysis

[a]SQL injection is one type of attack involving the so-called injection of a malicious piece of code. It is an attack on security by inserting a malicious SQL statement into an input field with the intention of executing it

When to Use Exploratory Testing?
Exploratory testing will be a good, effective, and efficient solution if one or more of the following premises are met:

- Specification of the product under test is incomplete, of poor quality, or lacking.
- There is time pressure; testers are short on time to conduct tests.
- Testers know the product well and are experienced in exploratory testing.

Exploratory testing is strongly related to the reactive test strategy (see Sect. 5.1.1). Exploratory testing can use other black-box, white-box, and experience-based techniques. No one can dictate to the tester how to conduct the session. If, for example, the tester finds it useful to create a state transition diagram that describes how the system works and then, in an exploratory session, designs test cases and executes them, they have the right to do so. This is an example of using scripted testing as part of exploratory testing.

Example The organization plans to test the basic functionality of a website. A decision has been made to conduct exploratory testing, and a test charter shown in Table 4.13 has been prepared for the tester. The test manager, based on the results collected from multiple exploratory test sessions, derives the following sample metrics for further analysis:

- Number of sessions conducted and completed
- Number of defects reported
- Time spent preparing for the session
- Time of the actual test session
- Time spent analyzing problems
- Number of functionalities covered

Exploratory Testing Tours
Andrew Whittaker [55] proposes a set of exploratory testing approaches inspired by sightseeing and tourism. This metaphor allows testers to increase their creativity when conducting exploratory sessions. The goals of these approaches are:

- To understand how the application works, what its interface looks like, what functionality it offers to the user
- To force the software to demonstrate its capabilities
- To find defects

The metaphor of the tourist allows to divide the software areas analogously to the parts of the city visited by the tourist:

- Business district—corresponds to those parts of the software that are related to its "business" part, i.e., the functionalities and features that the software offers to users.
- Historic district (historic center)—corresponds to the legacy code and history of defective functionality.
- Tourist district—corresponds to those parts of the software that attract new users (tourists), functions that an advanced user (city resident) is unlikely to use anymore.
- Entertainment district—corresponds to support functions and features related to usability and user interface.
- Hotel district—the place where the tourist rests; corresponds to the moments when the user does not actively use the software but the software still does its work.
- Suspicious neighborhoods—places where it is better not to venture; in the software, they correspond to places where various types of attacks can be launched against the application.

Whittaker describes a range of exploration (sightseeing) types for each district.

For more information on exploratory testing, see [56, 57].

4.4.3 Checklist-Based Testing

Description of the Technique

Checklist-based testing 🕮—like the previous two techniques described in this section—uses the tester's knowledge and experience, but the basis for test execution is the items contained in the so-called checklist. The checklist contains the test conditions to be verified. The checklist should not contain items that can be checked automatically, items that function better as entry/exit criteria, or items that are too general [58].

Checklist-based testing may seem like a technique similar to fault attacks. The difference between these techniques is that a fault attack starts from defects and failures, and the tester's action is to reveal issues, given a particular failure or defect. In checklist-based testing, the tester also acts in a systematic way but checks the "positive" features of the software. The test basis in this technique is the checklist itself.

Checklist items are often formulated in the form of a question. The checklist should allow you to check each of its items separately and directly. Items in the checklist may refer to requirements, quality characteristics, or other forms of test condition. Checklists can be created to support various test types, including functional and nonfunctional tests (e.g., 10 heuristics for usability testing [59]).

Some checklist items may gradually become less effective over time as those developing the checklist learn to avoid making the same errors. New items may also need to be added to reflect high-severity defects that have been recently discovered. Therefore, checklists should be updated regularly based on defect analysis. However, care should be taken that the checklist does not become too long [60].

In the absence of detailed test cases, testing based on checklists can provide a degree of consistency for testing. Two testers working with the same checklist will probably perform their task slightly differently, but in general, they will test the same things (the same test conditions), so necessarily their tests will be similar. If the checklists are high level, there is likely to be some variability in the actual testing, resulting in potentially higher coverage but less repeatability of tests.

Types of Checklists

There are many different types of checklists for different aspects of software. In addition, checklists can have varying degrees of generality and a narrower or broader field of application. For example, checklists for the use of code testing (e.g., in component testing) will tend to be very detailed and will usually include a lot of technical detail about aspects of code development in a particular programming language. In contrast, a checklist for usability testing may be very high level and general. Of course, this is not a rule—the testers should always adjust the level of detail in the checklist to suit their own needs.

Application

Checklists can be used for basically any type of testing. In particular, they can apply to functional and nonfunctional testing.

In checklist-based testing, the tester designs, implements, and runs tests to cover the test conditions found in the checklist. The testers can use existing checklists (e.g., available on the Internet) or can modify and adapt them to their needs. They can also create such lists themselves, based on their own and their organization's experience with defects and failures, their knowledge of the expectations of users of the developed product, or their knowledge of the causes and symptoms of software failures. Referring to one's own experience can make this technique more effective, because usually, the same people, in the same organization, working on similar products, will make similar errors. Typically, less experienced testers work on already existing checklists.

Coverage

For a checklist-based testing, no specific measures of coverage are defined. Of course, at the minimum, each checklist item should be covered. However, since it is difficult to know to what extent such an item has been covered (due to the fact that the technique appeals to the knowledge, experience, and intuition of the individual tester), this has both advantages and disadvantages.

The disadvantage, of course, is the lack of detailed coverage information and less repeatability than with formal techniques. The advantage, on the other hand, is that more coverage can be achieved if the same test list is used by two or more testers. This is because each of them will most likely perform a slightly different set of steps to cover the same checklist item. Thus, there will be some variability in the tests, which will translate into greater coverage but at the expense of less repeatability.

Example Below we present the two sample checklists. The first one contains the so-called Nielsen usability heuristics. Next to each checklist item, a comment is additionally provided in parentheses, which can help the tester with testing.

Nielsen heuristics for system usability

1. **Visibility of system status**: is the status of the system shown at every point in its operation? (The user should always know where he or she is; the system should use so-called breadcrumbs, showing the path the user has taken, as well as clear titles for each screen)
2. **Match between system and the real world**: Is there compatibility between the system and reality? (The application should not use technical language but simple language used every day by users of the system, relating to the business domain in which the program operates)
3. **User control and freedom**: does the user have control over the system? (e.g., it should be possible to undo an action that the user performed by mistake, such as removing a product put there by mistake from the shopping cart)
4. **Consistency and standards**: does the system maintain consistency and standards? (Appearance solutions should be consistent throughout the application, e.g., the same formatting of links, use of the same font; familiar approaches should also be used, e.g., placing the company logo in the upper left corner of the screen)

5. **Error prevention**: does the system adequately prevent errors? (The user should not get the impression that something has gone wrong; for example, we should not give the user the option to select a version of a product that is not available in the store, only to see a "no stock" error later).
6. **Recognition rather than recall**: does the system allow you to select instead of forcing you to remember? (e.g., descriptions of fields in a form should not disappear when you start filling them in—this happens when the description is initially placed inside that field)
7. **Flexibility and efficiency of use**: does the system provide flexibility and efficiency? (e.g., advanced search options should be hidden by default if most users do not use them)
8. **Aesthetic and minimalist design**: is the system aesthetically pleasing and not overloaded with content? (The application should appeal to users; it should have a good design, appropriate color scheme, layout of elements on the page, etc.).
9. **Help users recognize, diagnose, and recover from errors**: is effective error handling provided? (once an error occurs, the user should not receive a technical message about it or that it is the user's fault; there should also be information about what the user should do in this situation)
10. **Help and documentation**: is there help available on the system? Is there user documentation? (Some users will need a help function; such an option should be available in the application; it should also include contact information for technical support)

The second example of a checklist concerns code review. This list, as an example, includes only selected technical aspects of good code writing practices and is far from being complete.

Checklist for code inspections

1. Is the code written according to current standards?
2. Is the code formatted in a consistent manner?
3. Are there functions that are never called?
4. Are the implemented algorithms efficient, with appropriate computational complexity?
5. Is memory being used effectively?
6. Are there variables that are used without being declared first?
7. Is the code properly documented?
8. Is the manner of commenting consistent?
9. Is every divide operation protected against division by zero?
10. In IF-THEN statements, are the blocks of statements executed most often checked first?
11. Does each CASE statement have a default block?
12. Is any allocated memory released when it is no longer in use?

4.5 Collaboration-Based Test Approaches

FL-4.5.1 (K2) Explain how to write user stories in collaboration with developers and business representatives.
FL-4.5.2 (K2) Classify the different options for writing acceptance criteria.
FL-4.5.3 (K2) Use acceptance test-driven development (ATDD) to derive test cases.

The popularity of agile methodologies has led to the development of testing methods specific to this approach, taking into account the artifacts used by these methodologies and emphasizing collaboration between customers (business), developers, and testers. This chapter discusses the following issues related to testing in the context of agile methodologies, using a **collaboration-based test approach** 🕮:

- User stories as an agile counterpart to user requirements (Sect. 4.5.1)
- Acceptance criteria as the agile counterpart to test conditions, providing the basis for test design (Sect. 4.5.2)
- Acceptance test-driven development (ATDD) as a form of high-level testing (system testing and acceptance testing) often used in agile methodologies, based on a "test-first" approach (Sect. 4.5.3)

Each of the techniques described in Sects. 4.2, 4.3, and 4.4 has a specific goal with respect to detecting defects of a specific type. Collaborative approaches, on the other hand, focus also on avoiding defects through cooperation and communication.

4.5.1 Collaborative User Story Writing

In agile software development, a user story represents a functional increment that will be of value to the user, purchaser of the system or software, or any other stakeholder. User stories are written to capture requirements from the perspective of developers, testers, and business representatives. In sequential SDLC models, this shared vision of a specific software feature or function is achieved through formal reviews after the requirements have been written. In agile approaches, on the other hand, the shared vision is achieved either through frequent informal reviews during requirements writing or by writing requirements together, in a collaborative way, by testers, analysts, users, developers, and any other stakeholders. User stories consist of three aspects, known as the "3Cs:"

- Card
- Conversation
- Confirmation