Asynchronous programming in JavaScript can be challenging because it involves working with callbacks, Promises, and async/await, which can be difficult to reason about and can lead to bugs if not used correctly.

Here are some common issues that can arise when working with asynchronous code in JavaScript:

1. Callback Hell: This occurs when there are too many nested callback functions, making the code difficult to read and maintain. To avoid this, you can use Promises or async/await to make the code more readable.
2. Race Conditions: This occurs when two or more async operations depend on each other and their order matters. To avoid race conditions, you can use locks or semaphores.
3. Deadlocks: This occurs when two or more async operations are waiting on each other to complete, resulting in a standstill. To avoid deadlocks, you can use timeouts or cancelable Promises.
4. Incorrect Order of Execution: This can occur when async operations are not properly sequenced. To avoid this, you can use async/await or Promise.all().
5. Unhandled Rejections: This occurs when a rejected Promise is not properly handled with a catch block. To avoid this, you can use try/catch blocks or make sure to include a catch block for every rejected Promise.

It's important to thoroughly test your asynchronous code to ensure that it is functioning as intended and to catch any potential issues.

Here are some "code smells" that may indicate problems with asynchronous code in JavaScript:

1. Nested callback functions: Nested callback functions can make code difficult to read and maintain, and may indicate that the code could be refactored using Promises or async/await.
2. Callback functions with many arguments: Callback functions with many arguments may indicate that the code could be refactored to use Promises or async/await, which can make the code easier to read and understand.
3. Long chains of Promise.then() calls: Long chains of Promise.then() calls can make code difficult to read and maintain, and may indicate that the code could be refactored using async/await.
4. Lack of error handling: If a Promise is rejected and the rejection is not properly handled, this can lead to bugs and unexpected behavior in the code. Make sure to include catch blocks for every rejected Promise.
5. Repeated code blocks: If you find yourself repeating the same code block in multiple places, this may indicate that the code could be refactored to use async/await or Promises.
6. Complex logic in callback functions: If you find yourself writing complex logic in callback functions, this may indicate that the code could be refactored to use async/await or Promises, which can make the code easier to read and understand.

It's important to regularly review your code for these and other code smells, as they can indicate that the code could be improved in terms of readability, maintainability, and reliability.

Here are some hypotheses that could be tested related to asynchronous programming constructs in JavaScript and software quality:

1. Async/await results in more readable and maintainable code compared to other asynchronous programming constructs.
2. Promises have better performance compared to callbacks in JavaScript applications.
3. Async/await results in fewer bugs and errors compared to other asynchronous programming constructs.
4. Try/catch blocks are more effective at handling and recovering from errors compared to catch blocks for rejected Promises.
5. Async/await results in more concurrent code compared to other asynchronous programming constructs.
6. Promises result in more modular and reusable code compared to callbacks.
7. Cancelable Promises result in more fault-tolerant code compared to other asynchronous programming constructs.

These hypotheses could be tested through a combination of code review, testing, and performance analysis. It's important to carefully design and conduct experiments to ensure that the results are reliable and accurately reflect the impact of asynchronous programming constructs on software quality.

Functional correctness is a measure of how well a software system performs the tasks it was designed to do. A system that is functionally correct is able to perform all of its intended functions accurately and reliably.

In software quality, functional correctness is an important consideration because it determines whether a system is fit for its intended purpose. A system that is not functionally correct may produce incorrect results, fail to perform certain tasks, or behave unexpectedly, which can have serious consequences depending on the context in which the system is used.

To ensure functional correctness, it's important to thoroughly test and validate software systems to ensure that they are performing as intended. This can involve testing the system with a variety of inputs and conditions to ensure that it behaves correctly and produces accurate results.

Overall, functional correctness is an important aspect of software quality and is essential for ensuring that a system is reliable and fit for its intended purpose.

1. Introduction: This section could provide an overview of the topic and the research questions being addressed in the thesis. It could also include a background on asynchronous programming in JavaScript and the different constructs that are available for this purpose.
2. Literature Review: This section could review existing research on asynchronous programming in JavaScript and its relationship to software quality. It could also discuss related topics such as software engineering practices, code complexity, and test coverage.
3. Methodology: This section could describe the research design and methods that were used to conduct the repository mining study. This could include information on the sample of repositories that were selected, the data that was collected and analyzed, and the tools and techniques that were used.
4. Results: This section could present the findings of the study, including any trends or patterns that were identified in the data. It could also discuss any correlations or relationships that were found between the use of different asynchronous programming constructs and measures of software quality.
5. Discussion: This section could interpret the results of the study and discuss their implications for software development practices. It could also consider limitations of the study and suggest directions for future research.
6. Conclusion: This section could summarize the main findings of the study and provide a final conclusion on the research questions. It could also discuss any recommendations for developers or organizations that may be interested in using asynchronous programming constructs in their own projects.
7. References: This section could list all of the sources that were cited in the thesis, using a standardized citation style.
8. Appendices: This section could include any additional material that is relevant to the thesis but may not be necessary for the main text, such as tables or figures.