

Virtual Collaboration and Pair Programming

Pair programming is the de facto work method that most large engineering organizations use for “hands on keyboard” coding. Two developers, working synchronously, looking at the same screen and attempting to code and design together, which often results in better and clearer code than either could produce individually.

Pair programming works well under the correct circumstances, but it loses some of its charm when executed in a completely virtual setting. The virtual setup still involves two developers looking at the same screen and talking out their designs, but there are often logistical issues to deal with, including lag, microphone set up issues, workspace and personal considerations, and many other small, individually trivial problems that worsen the experience.

Virtual work patterns are different from the in-person patterns we are accustomed to. Pair programming at its core is based on the following principles:

1. Generating clarity through communication
2. Producing higher quality through collaboration
3. Creating ownership through equal contribution

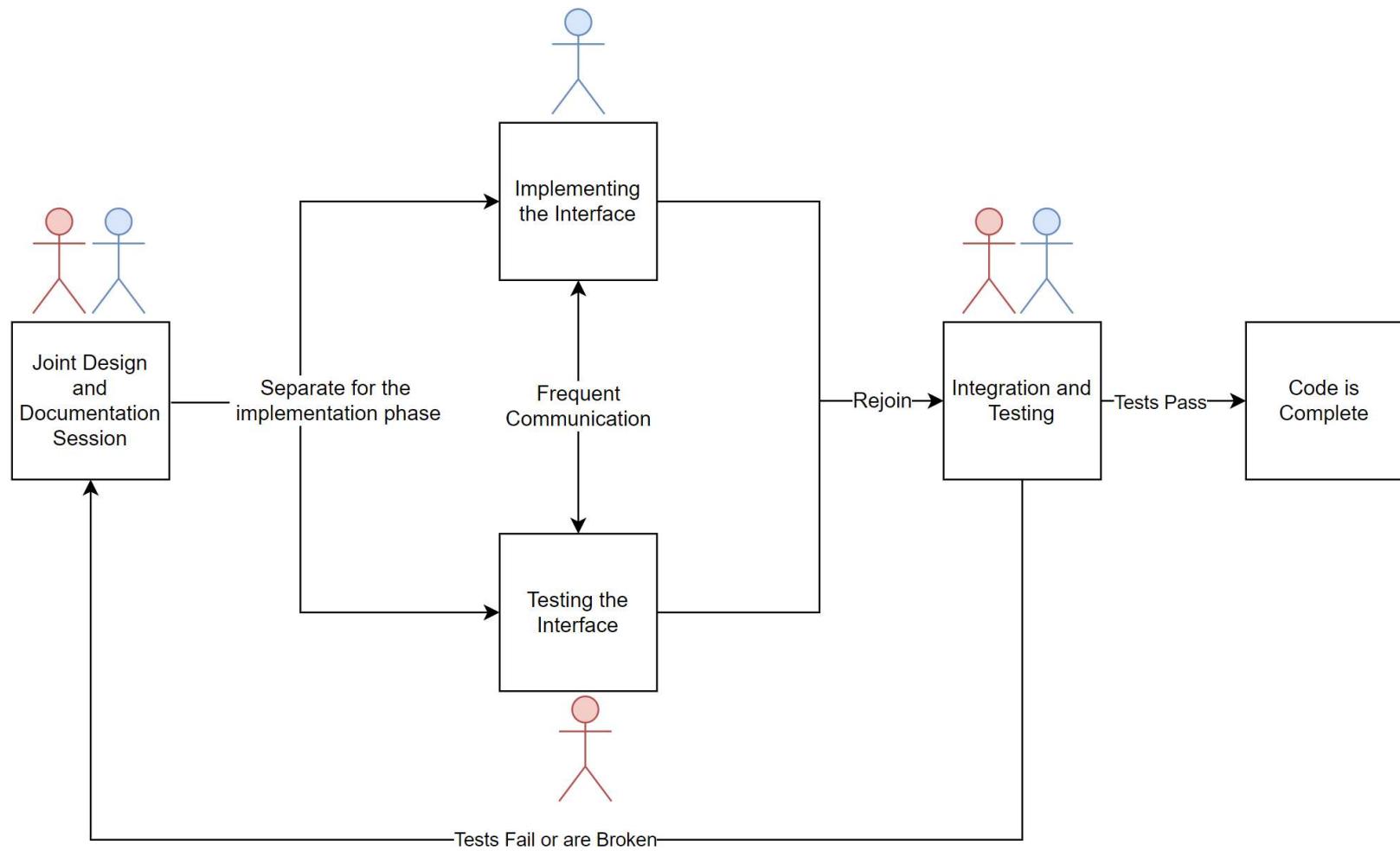
Pair programming is one way to achieve these results. Red Team Testing (RTT) is an alternate programming method that uses the same principles but with some of the advantages that virtual work methods provide.

Red Team Testing (RTT)

Red Team Testing borrows its name from the “Red Team” and “Blue Team” paradigm of penetration testing, and is a collaborative, parallel way of working virtually. In Red Team Testing, two developers jointly decide on the interface, architecture, and design of the program, and

then separate for the implementation phase. One developer writes tests using the public interface, attempting to perform edge case testing, input validation, and otherwise stress testing the interface. The second developer is simultaneously writing the implementation which will eventually be tested.

Red Team Testing has the same philosophy as any other Test-Driven Development lifecycle: All implementation is separated from the interface, and the interface can be tested with no knowledge of the implementation.



Steps

1. Design Phase: Both developers design the interface together. This includes:
 - Method signatures and names
 - Writing documentation or docstrings for what the methods are intended to do.
 - Architecture decisions that would influence testing (Factory patterns, etc.)
2. Implementation Phase: The developers separate and parallelize work, while continuing to communicate.
 - Developer A will design the implementation of the methods, adhering to the previously decided design.
 - Developer B will concurrently write tests for the same method signatures, without knowing details of the implementation.
3. Integration & Testing Phase: Both developers commit their code and run the tests.
 - Utopian Scenario: All tests run and pass correctly.
 - Realistic Scenario: The tests have either broken or failed due to flaws in testing. This leads to further clarification of the design and a discussion of why the tests failed.
4. The developers will repeat the three phases until the code is functional and tested.

When to Follow the RTT Strategy

RTT works well under specific circumstances. If collaboration needs to happen virtually, and all communication is virtual, RTT reduces the need for constant communication while maintaining the benefits of a joint design session. This considers the human element: Virtual communication is more exhausting than in person communication.

RTT also works well when there is complete consensus, or no consensus at all, on what purpose the code serves. Since creating the design jointly and agreeing to implement and test against it are part of the RTT method, RTT forcibly creates clarity through iteration and communication.

Benefits

RTT has many of the same benefits as Pair Programming and Test-Driven development but tries to update them for a virtual setting.

- Code implementation and testing can be done in parallel, over long distances or across time zones, which reduces the overall time taken to finish writing the code.

- RTT maintains the pair programming paradigm, while reducing the need for video communication or constant communication between developers.
- RTT allows detailed focus on design and engineering alignment before implementing any code, leading to cleaner and simpler interfaces.
- RTT encourages testing to be prioritized alongside implementation, instead of having testing follow or be influenced by the implementation of the code.
- Documentation is inherently a part of RTT, since both the implementer and the tester need correct, up to date documentation, in the implementation phase.

What You Need for RTT to Work Well

- Demand for constant communication and good teamwork may pose a challenge; daily updates amongst team members are essential to maintain alignment on varying code requirements.
- Clarity of the code design and testing strategy must be established beforehand and documented as reference. Lack of an established design will cause misalignment between the two major pieces of work and a need for time-consuming refactoring.
- RTT does not work well if only one developer has knowledge of the overall design. Team communication is critical to ensuring that every developer involved in RTT is on the same page.