Practical Experience with Test-Driven Development during Commissioning of the Multi-Star AO System ARGOS

M. Kulas[1], Jose Luis Borelli[1], Wolfgang Gässler[1], Diethard Peter[1], Sebastian Rabien[2], Gilles Orban de Xivry[2], Lorenzo Busoni[3], Marco Bonaglia[3], Tommaso Mazzoni[3], Gustavo Rahmer[4]

[1] Max Planck Institute for Astronomy, Königstuhl 17, Heidelberg, Germany
[2] Max Planck Institute for Extraterrestrial Physics, Giessenbachstraß, Garching, Germany
[3] INAF, Osservatorio Astrofisico di Arcetri, L.go E. Fermi 5, Firenze, Italy
[4] LBT Observatory, University of Arizona, 933 N. Cherry Ave, Tucson, Arizona, U.S.A.

ABSTRACT

Commissioning time for an instrument at an observatory is precious, especially the night time. Whenever astronomers come up with a software feature request or point out a software defect, the software engineers have the task to find a solution and implement it as fast as possible. In this project phase, the software engineers work under time pressure and stress to deliver a functional instrument control software (ICS). The shortness of development time during commissioning is a constraint for software engineering teams and applies to the ARGOS project as well. The goal of the ARGOS (Advanced Rayleigh guided Ground layer adaptive Optics System) project is the upgrade of the Large Binocular Telescope (LBT) with an adaptive optics (AO) system consisting of six Rayleigh laser guide stars and wavefront sensors. For developing the ICS, we used the technique Test-Driven Development (TDD) whose main rule demands that the programmer writes test code before production code. Thereby, TDD can yield a software system, that grows without defects and eases maintenance. Having applied TDD in a calm and relaxed environment like office and laboratory, the ARGOS team has profited from the benefits of TDD. Before the commissioning, we were worried that the time pressure in that tough project phase would force us to drop TDD because we would spend more time writing test code than it would be worth. Despite this concern at the beginning, we could keep TDD most of the time also in this project phase

This report describes the practical application and performance of TDD including its benefits, limitations and problems during the ARGOS commissioning. Furthermore, it covers our experience with pair programming and continuous integration at the telescope.

Keywords: Test-Driven Development, TDD, commissioning, continuous integration, pair programming, software engineering, testing, instrument control software, distributed software system, adaptive optics, LBT

1. SET UP

The goal of the multi-star AO system ARGOS is improving the image quality for the imager and spectrograph LUCI. In order to run a complex distributed hardware system like ARGOS, the telescope operators require assistance from the instrument control software (ICS) that the software engineers have to develop partly at the telescope on the mountain under the harsh conditions of an instrument commissioning.

1.1 Laser Guide Star System ARGOS

The Large Binocular Telescope (LBT) on Mt. Graham in Arizona consists of two primary mirrors each with a diameter of 8.4 meter. The already installed first light adaptive optics (FLAO) system provides diffraction limited images by using natural guide stars and deforming two adaptive secondary mirrors (ASM). The ARGOS project will also use these ASMs and it will operate additionally six green Rayleigh laser guide stars in order to correct the atmospheric turbulence near the ground.[1] By implementing a ground layer AO system, ARGOS will provide a corrected field of view of 4 arc minute diameter and it will increase the image quality by a factor of about 2-3 for the near-IR imager and multi-object spectrograph LUCI.[2] Especially, ARGOS will reduce the observation time for the same signal-to-noise ratio in the spectroscopy mode.

Software and Cyberinfrastructure for Astronomy III, edited by Gianluca Chiozzi, Nicole M. Radziwill,
Proc. of SPIE Vol. 9152, 91520D · © 2014 SPIE · CCC code: 0277-786X/14/$18
doi: 10.1117/12.2056218

Besides the six lasers, the ARGOS hardware equipment contains several other devices like calibration sources, slope computers, cameras and motors. The ARGOS hardware setup is complicated by the fact that the devices are distributed allover the telescope structure. For example, the laser boxes are located in the center of the telescope while the launch projector mirror sits on top of the ASM. Because there exists no central place for the ARGOS hardware, problems originating from many telescope locations can render ARGOS useless.

Other subsystems deal with the avoidance of illuminating aircraft and satellites. Human spotters watch for aircraft and they are empowered to stop the laser propagation remotely. A space command center approves for each observation target a list of time slots in which it permits to propagate lasers. On ARGOS side, a satellite avoidance service compares the current telescope position and time of day with the permitted time slots. If the current time of day is missing in the list of allowed time slots, it forbids the laser propagation by trigger a signal to the laser interlock.

The Argos project started in 2006 with consortium members from Germany, Italy and the USA. In early 2010 then, ARGOS team finished the final design report (FDR), propagated the first green laser beams on sky in autumn 2013 and closed the AO loop for the first time in May 2014.

1.2 Overview of ARGOS Instrument Control Software

Today's astronomical instruments are software-intensive. The software programs like the instrument control software (ICS) for ARGOS automatizes complex procedures, performs complicated computations quickly and provides flexibility to their users like e.g. allowing parameter adjustments. The ICS is a program package for carrying out repetitive, boring and error-prone tasks during observations. By this assistance, humans are able to utilize the telescope effectively.

Nowadays, control software of astronomical instruments takes care of a large number of components and subunits. In such software systems, the huge amount of components and their multitudinous dependencies among each other cause complexity. People have problems to understand and predict the behaviour of complex systems. To make things worse, the ICS is a discrete system whose behaviour is difficult to describe in terms of continuous mathematics.[3]

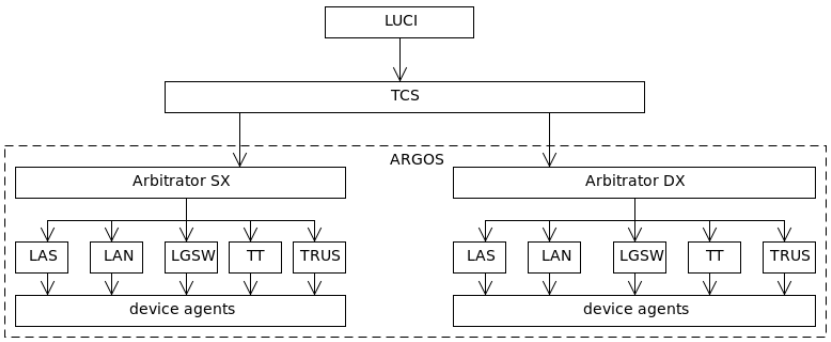

Figure 1. Embedding of the ARGOS instrument control software in the context of the telescope control system (TCS) and the instrument LUCI. The ARGOS arbitrator of the two telescope sides DX (right) and SX (left) commands the five subsystems LAN, LAS and LGSW, TT and TRUS. The countless device agents provide abstractions from the actual hardware.

The ARGOS way for mastering the ICS complexity is applying well-established software-engineering techniques and a simple architecture based on principles of service-oriented architectures.[4] An autonomous device agent ("basdard") is assigned for each device, so that users operate the individual devices via consistent GUIs and scripting interfaces. Amongst other things, it is the responsibility of the agents to tolerate device failures. Using these agents, the ARGOS subsystems close nine control loops to allow a smooth AO operation. The arbitrator accepts commands from the instrument LUCI via the telescope control system (TCS) and provides a simplified interface to the subsystems. This layered architecture demonstrates a hierarchical structure with a separation of concerns.

1.3 Theoretical Benefits of TDD

Software programs are brittle products. Flipping just one bit in a multi-megabyte large program, can break the behaviour of the software. Compared to buildings, removing one brick causes no crash. It is the responsibility of the software developer to ensure the correctness of his program. That means that the developer has to test the proper functioning of their programs. But manual testing is slow and tedious. A better solution is using the computer for automatic testing of programs.

Test-driven development (TDD) is a software development technique that requires the programmers to write a failing test before writing any production code as a fix for the failed test.[5] Applying this technique strictly, the programmers gain automatically running tests with complete test code coverage. That means that passing tests tell them that their programs fulfills the desired behaviour. By running the tests frequently, the programmers get rapid feedback about defects. Thus, the production code is entangled with test code. The following list describes the the benefits of TDD:

1) The automatic executable tests are a kind of ***safety net*** for the software developers. When refactoring, fixing defects or adding features, the fear to break existing code is gone because these tests will tell immediately the success of such source code surgeries.

2) A well-written test suite eases the ***maintenance*** of large software systems like the ARGOS ICS. When the original author of such a system has left the project, source code becomes legacy code when tests are missing. The subsequent programmers who are in charge of such code base often find it difficult to apply code changes because of the uncertainty about the effects of their actions. These source code maintainers appreciate the availability of automatic tests that expose broken functionality immediately.

3) TDD can act as a ***design aid***. Writing the tests first forces the developers to think about the design and the implementation of software module interfaces. The fact that the production code must be testable leads often to simple designs. Thus, the tests can guide the software developers. For instance, the design of all ARGOS subsystem controllers was driven by tests.

4) Tests are a kind of ***executable specification***. Many software projects lack a precise specification of the required software functionality. In the ARGOS ICS, the test cases have meaningful names for describing the behaviour of the software modules. Thereby, the tests serves as a specification.

5) Documenting software systems is often neglected because creating and updating ***documentation*** is time-consuming. This problem can be solved by automatic test suites. When consulting the tests, software engineers learn the programming interface by examples. As an additional benefit, this documentation is expressed in a formal language, namely a programming language, and therefore executable. However, this is no excuse to the software engineer to skip documentation for the user.

6) Writing production code and test code simultaneously allows to keep up the ***productivity velocity***. A good test suite detects regression and allows the developer to fix defects without breaking existing behaviour. The initial cost of setting up the test framework is negligible compared with the obtained productivity. The test suite of the ARGOS ICS lets the software engineers concentrate on new features instead of worrying about breaking existing code.

7) TDD is a ***mature technique***. It is a key element in agile development methods like Extreme Programming (XP).[6] Many frameworks assist the software engineering team to write unit tests and modern integrated development environments (IDE) have built-in support for TDD. For instance, we use the IDE Eclipse[7] with the PyDev plugin[8] to test drive the implementation of Python code.

8) Writing a test case forces the software engineer to concentrate at this specific feature and supports to do only one thing at a time. This property of TDD ***avoids distractions*** and keeps the source code development on track.[9]

TDD is not a silver bullet and comes with downsides. For instance, it is easy to write fragile tests, i.e. a test fails when it should not fail. Or the test cases are so complex that they become unmaintainable. As a result, they are discarded. This is a sign that the programming team lacks TDD experience. The writing of a test case requires discipline, especially when the implementation appears obvious and riskless to implement.

1.4 Software Engineering Team

Developing a large software system is team work. The construction of the ARGOS ICS is no exception. The software engineering group sits in Heidelberg and consists of two full-time programmers with a educational background in computer science and one software project manager who acts as the the domain expert for AO for the programmers. Beside this core software engineering team, the hardware engineers of the other consortium partners are responsible for constructing the subsystem hardware and are also experienced with writing control software for astronomical instruments. Due to their detailed knowledge of the subsystem hardware, these hardware engineers have been predestined to contribute to the ARGOS ICS in form of providing central algorithms. For instance, our Italian consortium partner has built the wavefront sensor (WFS) and therefore, It has been most effective for the whole project that this partner has implemented the WFS control procedures and algorithms. In this case, the Italian hardware engineers have become part of the ARGOS software engineering team. The programmers from Heidelberg have supported them by providing a software framework and teaching TDD.

The team of the project partners are mainly physicists without any experience of TDD but with interest in this development technique especially after they realized the benefits of TDD in the ARGOS ICS. In previous projects, the ICS was the part of the instrument that many project partners have complained about, e.g. due to the high number of defects. Being aware of the trouble with previous ICS, they were open minded and curious to learn the basics of TDD in order to contribute to the ARGOS ICS.

The project management lacked experience with TDD. That is why the management forced no development technique, i.e. the project managers neither recommended TDD nor prohibited TDD. Instead they trusted the software engineers and allowed them to organize themselves. If the software development team is disciplined and competent, this approach works.

1.5 Development platform

The right software tools speed up the development of a software product. With a hammer and nails, the construction of a small wooden hut is feasible. But it is unlikely that building a skyscraper with these tools will succeed. This also applies to the field of software engineering where tools assist the engineer to master complexity.

Development platforms which are open source supports the spreading of TDD in the ARGOS software team. The chosen IDE was Eclipse[7] with plugins for C++ and PyDev.[8] The usage of many 3rd party libraries like numpy[10] helped to keep the source code slim. For writing Python based tests, the we used the Python module `unittest` and for C++ the Boost test library.[11] At end of May 2014, the production code of the ARGOS ICS consisted of about 310 C++ files containing around 32,000 lines of code and up to 400 Python files with ca. 56,000 lines of code.

The size of the instrument software source code made it nearly impossible for any human to predict the effect of a change in the code base or in the 3rd party libraries. Therefore, we configured the continuous integration server Jenkins[12] to check our source code repository for changes. In case of a change set, Jenkins checked out a fresh working copy of the ARGOS ICS, built it reliably and run all tests in a known environment on the target operating system CentOS 6. Often we had the impression that Jenkins was an additional person of the team because it reported many integration failures like missing entries in configuration files of Automake.[13]

1.6 Software Engineering Challenges during Commissioning

During commissioning, the ARGOS team installs the instrument for the first time at the telescope. In this phase the electronic engineers, mechanical engineers, software engineers and scientists collaborate at the same time with the goal to integrate the instrument into the telescope environment. Almost always, the programmers must adjust the ICS in order to make it work at the telescope, e.g. we figured out that some ports of an I/O device

was changed by the electronic engineer with the installation. Commissioning is the first time when users operate the instrument at its final location and often request changes or features in the ICS, e.g. the users wishes more details about the status of the subsystems.

Before the commissioning at the telescope, we assembled, integrated and tested the subsystems in the laboratory, therefore the name assembly, integration and testing phase (AIT). Compared with commissioning, the AIT took place in calm offices or laboratories and provided a relaxed and productive work environment in which we wrote the core of the ICS and prepared the software package in such a way to extend and to modify it quickly. The continuous integration with Jenkins worked well and we programmed many production components as a programming pair. During AIT, we applied TDD and delivered successfully the components of the ICS in an iterative and incremental manner.[14]

The LBT is built for night observations. It is necessary that part of the commissioning occurs in these precious nights, e.g. for final adjustments. Thus, night commissionings inhibits astronomical observations. Considering the funding of a telescope, one can say that night time is money because the telescope can only produce scientific results in this time of the day. Therefore night time for commissioning is limited and the engineers of all fields work hard to achieve as much as possible during these nights. This means for the software engineers that they have to finish feature requests as fast as possible and deliver the instrument software in time.

Of course, we had deadlines during AIT. But the conditions of commissioning required us to deliver software features immediately. Whenever a defect popped up during the commissioning, it was our task to provide working software for the engineering team. In such moments, we had a license to hack and to improvise. Nobody forced us to apply TDD. When we fixed defects or implemented features, some engineers were often waiting behind our back impatiently and were watching our programming work including all our cursor movements and typing errors. Most of the time, we worked in the telescope control room below the telescope dome in an altitude of about 3,200 meter. After a while we got used to the low oxygen level, and a working time of ten hours and more was not uncommon. The telescope control room was often crowded by technicians, engineers and astronomers which were discussing the next operating steps. In this noisy control room, we sat side by side in front of our laptops on the desk. Even tourist groups appeared casually. Due to the people and computer equipment, it sometimes was so hot inside the control room that we had to switch on an external fan. Especially in a domain like software engineering where everything happens in mind, more convenient and more productive working places are conceivable than the telescope control room during a commissioning run.

2. PERFORMANCE OF TDD DURING COMMISSIONING

2.1 Development of Production Code

The truth of a software system is its source code. No matter what any salesman tell, a look at the source code reveals its capabilities and limitations. Often a quick look at the source code files scares the reader if he is confronted with a complicated design, monster methods, variables with meaningless names and out-dated comments. A source code reader might even panic if it is his job to maintain such legacy code. We emphasize readable source code. For instance, we prefer choosing meaningful variable names instead of spending time explaining the meaning of the cryptic variable in a comment. Modern IDE facilitate later renaming of classes, functions or variables. We often found a better name for a class or variable and used the assistance of the IDE for performing a refactoring. In the middle of a commissioning night when we were tired but are forced to implement a feature, we appreciated such reliable help.

Up to now we have gained experience with TDD in five commissioning runs of about fifty days in total. Nearly all our writing of production code started with a writing of a failing test. Thus, we adhered to the basic rule of TDD. During AIT, we become proficient to use TDD and considered it as training for test-driving production code for the commissioning. However, there were the following situations during commissioning in which we wrote production code without test:

(a) The most effective way for the team to learn the behaviour of the installed instrument was experimenting with it. This was due to unexpected behaviour of the hardware and its effect on the algorithms. For instance, it was unclear which shape the laser spots would have on the patrol cameras. But the algorithm for finding the laser spots on the patrol camera was depending on the appearance of these laser spots.

(b) A team member found a severe flaw in the ICS that stopped him to finish his task. In this case, we provided hot fixes. For example, the controller for the launch system refused to move the launch mirror due to an insane telescope pointing position.

In these situations, we wrote production code without any test. This action violated the basic TDD rule but it was mitigated by the fact that we were able to check this new production code immediately against the instrument. Fortunately, these emergency situations were rare and the testless production code changes left the software architecture untouched. These situations are similar when writing GUIs where tinkering and manual exploration is more economic than crafting tests that check the visual appearance. Anyway, the focus of TDD is not about testing every execution path but to provide confidence to the software engineer that the production code is ready for deployment. Actually, testing the ICS is a task for testers but the ARGOS project like almost all astronomical instrumentation projects lacks this kind of quality assurance. As a result, the users are finally the testers. Our development style was a tradeoff between being effective in the telescope control room and ensuring the functionality with tests.

The ARGOS ICS comes with many unit tests and several integration tests (end to end test). Almost every class has its unit test which cover its functionality. Compared with unit tests, the setup for integration tests requires more effort. Integration tests are end to end tests and their goal is to check the interfacing of the production modules and basic functionality. For instance, our integration tests checked the wiring of the subsystem controller logic with the middleware.

In unit tests we often began with implementing the optimistic test cases, a.k.a happy paths. Then we considered the realistic test cases to cover major error cases. From our experience, the proper dealing with error cases took more time than the optimistic test cases. During AIT, we created most test suites. At the telescope, it was intuitive to start with a test case for a new feature because the test suite was already available and prepared for extension.

TDD often led us to designs which permitted checking functionality close to the hardware. We found it easy to test-drive the implementation of a motor driver library. By injecting a fake communication channel into the motor driver library, we were able to test various error conditions.

Although we team skipped some tests for production code in the rush of commissioning, we strove to cover all production code in test cases. This aim is mandatory especially in scripting languages like Python where semantic errors are only detected during run-time one by one. In a compiled language like C++, the compiler checks the semantics during translation.

Usually about two hours before the commissioning night, we deployed the ICS on the operation workstations. This installed ICS was a good starting point for the night in which the ARGOS team concentrated on understanding the whole instrument. During these commissioning nights we only fixed urgent defects that would have stopped the night work on the operation workstations. Developing the ICS on the instrument happened rarely in the night time because that time was too precious.

In unit tests, the object under test often collaborates with other objects. We faked the collaborating objects by creating test doubles, more precisely test spies[9] which record their interaction. Then we injected these test spies into the object under test. But writing the test spies was time-consuming. Therefore, we used the library mockito-python[15] to create test spies for Python unit tests. For C++, we wrote a custom test spy framework. From our gained experience, the intensive usage of test spies was problematic because loading these test doubles with a decent default behaviour in each test setup code was tedious. A well-behaving simulator would have facilitated the writing of the test setup. For instance, a flexure compensation algorithm object required a camera object for setting the binning and providing frames. When using a test spy, the programmer has to load the camera object to return the frames with the correct binning. In this case, a camera simulator object could perform the binning on the frame by its own and could simplify the test setup code.

2.2 Example: TDD for a Motor Driver with Python and Mockito

ARGOS uses motors for moving a dichroic mirror into the laser beams. Communication with the motors goes over a serial line that a terminal port server attaches to the telescope Ethernet network. The motors are daisy-chained;

therefore every motor has its own address. The first step in the unit test is to hide away the communication channel details and concentrate on the motor command set:

communication_channel.py (production code)

```
from abc import abstractmethod

class CommunicationChannel(object):
    """Abstract base class of a communication channel.

    Every command that is sent returns an answer.
    """
    @abstractmethod
    def sendAndWaitForAnswer(self, command):
        assert False
```

The first task is the implementation of the motor position query. The manual of the motor tells us that command has the format `<drive address> QA` and a valid answer is formatted according to the pattern `<drive address>QA,<position>`. With that knowledge of the manual, we write first the optimistic test case. For faking the communication channel, we mock it with python-mockito.[15]

dichroic_motor_test.py (optimistic case)

```
import unittest

from mockito import mock, when

from communication_channel import CommunicationChannel
from dichroic_motor import DichroicMotor

class DichroicMotorTest(unittest.TestCase):

    def test_optimistic_position_read_back(self):
        # mock() creates a test spy for a CommunicationChannel.
        channel= mock(CommunicationChannel)
        motor= DichroicMotor(driveAddr= "07", communicationChannel= channel)
        # when() loads an answer for the command "07QA".
        when(channel).sendAndWaitForAnswer("07QA").thenReturn("07QA,+19")

        self.assertEqual(19, motor.getPositionInSteps())

if __name__ == "__main__":
    unittest.main()
```

This test fails. In the next step, we solve the test failure by implementing a simple solution:

dichroic_motor.py (production code)

```
class DichroicMotor(object):

    def __init__(self, driveAddr, communicationChannel):
        self._driveAddr= driveAddr
        self._channel= communicationChannel

    def getPositionInSteps(self):
        answer= self._channel.sendAndWaitForAnswer(self._driveAddr + "QA")
        return int(answer.split(',')[1])
```

The next step is covering the error case in which the motor driver receives an unexpected answer. Before we concentrate on this test case, we realize that the upcoming test case requires the same test setup as the optimistic test case. We avoid the impulse to copy and paste the setup code. Instead we extract it into a common test fixture. Here is the dichroic motor test after the refactoring:

`dichroic_motor_test.py` (after refactoring)

```
class DichroicMotorTest(unittest.TestCase):

    def setUp(self):
        """Creates the test fixture.

        Every test case has access to the channel and motor object.
        """

        self.channel= mock(CommunicationChannel)
        self.motor= DichroicMotor(driveAddr= "07",
                                  communicationChannel= self.channel)

    def test_optimistic_position_read_back(self):
        when(self.channel).sendAndWaitForAnswer("07QA").thenReturn("07QA,+19")
        self.assertEqual(19, self.motor.getPositionInSteps())
```

Now, it is easy to add a failure scenario:

`dichroic_motor_test.py` (one pessimistic case)

```
    def test_complains_about_invalid_position(self):
        when(self.channel).sendAndWaitForAnswer("07QA").thenReturn("foobar")
        self.assertRaises(DichroicMotorError,
                          self.motor.getPositionInSteps)
```

Of course, this test case fails. The following simple implementation clears that test failure:

`dichroic_motor.py` (production code)

```
class DichroicMotorError(Exception):
    pass

class DichroicMotor(object):

    def __init__(self, driveAddr, communicationChannel):
        self._driveAddr= driveAddr
        self._channel= communicationChannel

    def getPositionInSteps(self):
        answer= self._channel.sendAndWaitForAnswer(self._driveAddr + "QA")
        try:
            return int(answer.split(',')[1])
        except:
            raise DichroicMotorError("Invalid␣position␣answer:␣%s" % answer)
```

So far, the dichroic motor implementation abstracts from the message formats and is independent of the communication channel. This test scaffolding facilitates the writing of more test cases like moving to the home position or performing an emergency stop.

2.3 Pair Programming

Developing instrument control software is a demanding and creative task. Software engineers often need assistance for this task. Due to today's complexity of software systems, the time of lonely code cowboys is over. Pair programming is an activity with two programmers sitting in front of one computer. While one programmer writes code, his partner asks questions, corrects him and comments on the written source code. This assistance requires that the partner concentrates the whole time on the work of the programming colleague. The roles between the two developers switch often.

During AIT, pair programming worked great for us. The quality of the source code was high and the knowledge about the ICS internals became widespread. Based on this good experience, we took over this habit into the commissioning runs. Depending on the difficulty of the engineering task, a programming pair formed automatically; each programmer decided by himself when a programming partner was necessary. A pair programming session was also an effective trick to fight against tiredness in the middle of the night: one programmer wrote the test code and his partner solved the failing test with production code. This rule resulted in a kind of game that kept both programmers awake.

The ARGOS ICS provides an interactive command interpreter named `argos_terminal` that allows engineers to access all services via a Python interface. With this tool, the software engineers had access to all hardware devices and were able to create algorithms quickly and conduct experiments with the instrument. Often more than two engineers worked together at one `argos_terminal` whereby a lively and productive team programming session arose.

2.4 Continuous Integration

Large software systems like the ARGOS ICS depend on many modules for their proper functioning. Whenever a developer commits a change set into the source code repository, the risk is high that the existing functionality breaks for the following reasons:

a) The change set modifies the behaviour of a internal module but other internal modules relies on the previous behaviour.

b) The application programming interface (API) of an external module like a 3rd party library on the target computer differs from the computer where the the software engineer has tested his production code.

One of our goals was the delivery of features as fast as possible with the constraint to keep existing functionality and to avoid regression. The automatic tests checked the availability of working features. Whenever a test complained, the integration of a change set failed, and the software engineer is obliged to update the test code or production code. This rule guaranteed to have a deployable ICS available all the time. But performing the integration task continuously by hand, was tedious. That is why, we set up the continuous integration server Jenkins[12] that built the whole ARGOS ICS and checked every change set that entered the source code repository

ARGOS ICS runs on the novel middleware TwiceAsNice (TaN).[16] We decided to go with the latest version of TaN because it contained bug fixes and improvements. Like any real-world software, the latest version of TaN often failed during compilation or sometimes broke backwards compatibility. For commissioning, we relied on a working version of TaN. Therefore, we configured Jenkins to watch the TaN source code repository and to build the ICS always with the latest version of TaN. Jenkins warned us about build failures regularly and our test suites from TDD casually reported regressions with TaN. In this way we were able to release our ICS without concerns about TaN.

During commissioning many engineers on several workstations with different versions of working copies were committing change sets. In this situation it was helpful that Jenkins continuously checked the integration of all code changes. However in hectic situations, we postponed fixing complaints of Jenkins when they concerned parts of the source code which were not relevant in that moment for commission. If the software engineering team has a pragmatic and undogmatic attitude, then it concentrates on the high priority tasks available.

3. TEAR DOWN

The tough conditions during the commissioning of ARGOS at the telescope constrained our software engineering job. Nevertheless, we applied test-driven development as often as possible which was one reason for the successful delivery of the instrument control software. Another reason was the continuous building of the software system and the execution of its test suites after each change set. This ensured that proper integration of all software modules. Furthermore, programming with one or more partners increased the code quality and spread knowledge about the internals of the ICS.

We were pleased by the results of applying TDD although we have decided to drop it in situations in which experimenting with the production code has promised to be more efficient than writing tests. This pragmatic attitude has allowed us to insert hot fixes and experimental code. At the end, we could provide a working ICS with a low number of defects and we are looking forward to apply TDD during our next commissioning runs.

REFERENCES

[1] Rabien, S., Ageorges, N., Barl, L., Beckmann, U., Blumchen, T., Bonaglia, M., Borelli, J. L., Brynnel, J., Busoni, L., Carbonaro, L., Davies, R., Deysenroth, M., Durney, O., Elberich, M., Esposito, S., Gasho, V., Gassler, W., Gemperlein, H., Genzel, R., Green, R., Haug, M., Hart, M. L., Hubbard, P., Kanneganti, S., Masciadri, E., Noenickx, J., de Xivry, G. O., Peter, D., Quirrenbach, A., Rademacher, M., Rix, H. W., Salinari, P., Schwab, C., Storm, J., Struder, L., Thiel, M., Weigelt, G., and Ziegleder, J., "ARGOS: the laser guide star system for the LBT," *Adaptive Optics Systems II* **7736**(1), 77360E, SPIE (2010).

[2] Hart, M., Rabien, S., Busoni, L., Barl, L., Beckmann, U., Bonaglia, M., Boose, Y., Borelli, J. L., Bluemchen, T., Carbonaro, L., Connot, C., Deysenroth, M., Davies, R., Durney, O., Elberich, M., Ertl, T., Esposito, S., Gaessler, W., Gasho, V., Gemperlein, H., Hubbard, P., Kanneganti, S., Kulas, M., Newman, K., Noenickx, J., de Xivry, G. O., Peter, D., Quirrenbach, A., Rademacher, M., Schwab, C., Storm, J., Vaitheeswaran, V., Weigelt, G., and Ziegleder, J., "Status report on the Large Binocular Telescope's ARGOS ground-layer AO system," *Astronomical Adaptive Optics Systems and Applications IV* **8149**(1), 81490J, SPIE (2011).

[3] Booch, G., Maksimchuk, R. A., Engle, M. W., Ph.D., B. J. Y., Conallen, J., and Houston, K. A., [*Object-Oriented Analysis and Design with Applications*], Addison-Wesley Professional, Boston, Massachusetts (2007).

[4] Borelli, J., Barl, L., Gässler, W., Kulas, M., and Rabien, S., "Service-oriented architecture for the ARGOS instrument control software," **8451**(88), SPIE (2012).

[5] Beck, K., [*Test Driven Development: By Example*], Addison-Wesley Professional, Boston, Massachusetts (2002).

[6] Beck, K. and Andres, C., [*Extreme Programming Explained: Embrace Change*], Addison-Wesley (2004).

[7] "Eclipse IDE." `http://eclipse.org/` (Retrieved 2012-06-06).

[8] "PyDev: Python plugin for Eclipse." `http://pydev.org/` (Retrieved 2012-06-06).

[9] Langr, J., [*Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better*], The Pragmatic Programmers (2013).

[10] "NumPy." `http://www.numpy.org/` (Retrieved 2014-06-02).

[11] "Boost." `http://www.boost.org/` (Retrieved 2014-06-02).

[12] "Jenkins." `http://jenkins-ci.org/` (Retrieved 2012-06-06).

[13] "GNU Automake." `http://http://www.gnu.org/software/automake/` (Retrieved 2014-06-06).

[14] Kulas, M., Barl, L., Borelli, J. L., Gässler, W., and Rabien, S., "Instrument control software development process for the multi-star AO system ARGOS," *Proc. SPIE* **8451**, 845109–845109–7 (2012).

[15] "mockito-python." `https://code.google.com/p/mockito-python/` (Retrieved 2014-06-02).

[16] "TwiceAsNice." `https://svn.mpia.de/trac/gulli/TwiceAsNice` (Retrieved 2012-06-06).