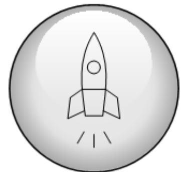
ICONIX PROCESS ROADMAPS

DOUG ROSENBERG



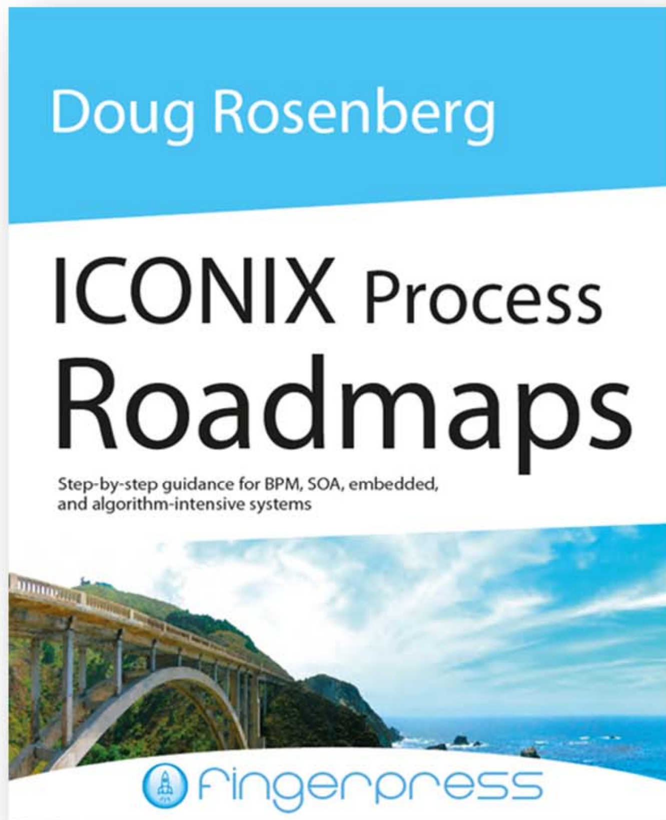
FINGERPRESS LTD
LONDON

Find more great books at:
www.fingerpress.co.uk

This PDF contains Appendix A of ***ICONIX Process Roadmaps*** *by Doug Rosenberg.*

Find out more at:
http://software.fingerpress.co.uk/iconix-process-roadmaps.html

Or buy the book direct:
www.iconix.org/product.sc?productId=92&categoryId=18



Copyright © Doug Rosenberg 2011. We encourage you to forward this PDF around and redistribute it for non-commercial use. We just ask that you don't modify the PDF; and if you like it, to buy a copy of the book!

APPENDIX A

Bonus Roadmap: Design Driven Testing for Systems

By Doug Rosenberg – many thanks to Michael Sievers at the Jet Propulsion Laboratory (Pasadena, CA) for his valuable inputs.

Design Driven Testing (DDT) was first outlined in the book *Use Case Driven Object Modeling with UML: Theory and Practice* (by Doug Rosenberg and Matt Stephens), and then described in detail in *Design Driven Testing: Test Smarter, Not Harder* by the same authors. Although DDT can be successfully used in conjunction with Test Driven Development (TDD), it really follows the opposite development philosophy: starting with a design (and requirements and use cases) and driving tests from them.

DDT is a highly methodical approach to testing, allowing you to know when you've finished – i.e. when you've written enough tests to cover the design and the requirements. It helps you to "zoom in" and write algorithmic tests to cover intensive or mission-critical sections of code, and to know when it's safe to "zoom out" and write fewer tests for boilerplate or less critical code. Consequently you tend to end up with fewer tests overall than with TDD – but comparatively more test *scenarios* per test case.

In this Appendix, Doug extends the concept of DDT for hardware/software systems, allowing SysML-based designs to be tested in a highly rigorous, systematic way. It's still Design Driven Testing, but now the design elements that need to be tested include all of the "four pillars of SysML", whereas DDT for software focuses on testing behavior.

Doug (not being an actual rocket scientist) would like to acknowledge some informal but very valuable and entertaining discussions with Dr. Michael Sievers of NASA's Jet Propulsion Laboratory (JPL) regarding how real rocket scientists would approach the problem of detecting potholes from outer space – and how they would test their designs.

Introduction

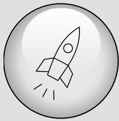
This Appendix discusses an integrated, model-based design approach based on an extension of Design Driven Testing (DDT)[54] that is compatible with Validation & Verification (V&V) activities at all design levels. **(We define V&V in the sidebar on Page 216)**. Models are validated at each step and the validated models are then used for verification tests. Essentially the design is the model and the model is the design. At some level of detail the "model" is physical hardware and executable software and the V&V process uses familiar unit, bench, and integration tests.

While DDT is related to software simulations currently used in the industry, it differs in at least one key aspect: typically software simulations are built by independent teams from independent requirements. There is always a question of how close the software simulation is to the real system. In our approach, there is one system model which may be viewed in different ways and at different levels of detail as suits a user's needs. Moreover, as will be seen in the Design Driven Testing section, our approach guides "smart" selection of test cases which saves cost and schedule. This is particularly important because while systems have become more complex, the time allocated for testing has not changed. We must now test more, but in the same amount of time as simpler systems. This can only become practical by improving test efficiency.

DDT is intimately linked to modern, model-based systems engineering using the SysML paradigm. Like SysML which establishes a single, common source for design truth, DDT provides a single, consistent source of V&V truth and establishes clear lines of responsibility. Equally important is that DDT is integrated with the iterative design process assuring that each level of design is consistent with its parent and children.

---

[54] *Design Driven Testing: Test Smarter, Not Harder* by Matt Stephens and Doug Rosenberg (Apress, 2010).



ROCKET SCIENCE: Why Use a Space Mission to Illustrate DDT/Systems?

A space mission is a system of interacting systems that implement a set of behaviors defined by operational concepts. Operations concepts describe all phases of the mission from ground-based testing through pre-launch, launch, separation, commissioning, nominal operations and eventually disposal or decommissioning. Each of these "use cases" comprises complex actions, alternate paths and exceptional conditions. When a typical mission costs taxpayers hundreds of millions to billions of dollars, it is crucial that these systems are validated and verified from the highest conceptual levels down to the detailed code that defines the functions in an FPGA. The cost of finding and fixing problems escalates by orders of magnitude as the mission progresses through its design and deployment stages hence, systematic and early Validation and Verification (V&V) are central aspects of all space programs.

A Model System Design

Using SysML, we describe the design of a fanciful, yet somewhat realistic spacecraft mission for a running example in this paper. The next section discusses the system V&V process as an extension to DDT. We then illustrate the system V&V method applied to our spacecraft design.

Our mission goal is a system that images the city streets it passes over with a sufficiently large telescope that it can resolve large street imperfections. We call our spacecraft: Persistent Open Traffic Hazard Obstacles, Large and Elongated (POTHOLE) because all spacecraft must have a clever name. POTHOLE sends images to the ground where they are processed and new or growing potholes can be reported to local maintenance bureaus.

Figure A-1 shows a conceptual diagram of POTHOLE spacecraft. Major visible components are the solar array in green, telescope in dark blue, sunshade in light blue, downlink antenna in silver and the "Bus" in yellow. The Bus consists of the functions needed to maintain the vehicle and communicate with the ground. The telescope and related imaging functions are collectively called the "Instrument." The mission block diagram in Figure 3 shows that the POTHOLE spacecraft is a composition of the Bus and Instrument systems. The mission also requires a Ground Station for controlling the spacecraft and accepting and processing images.

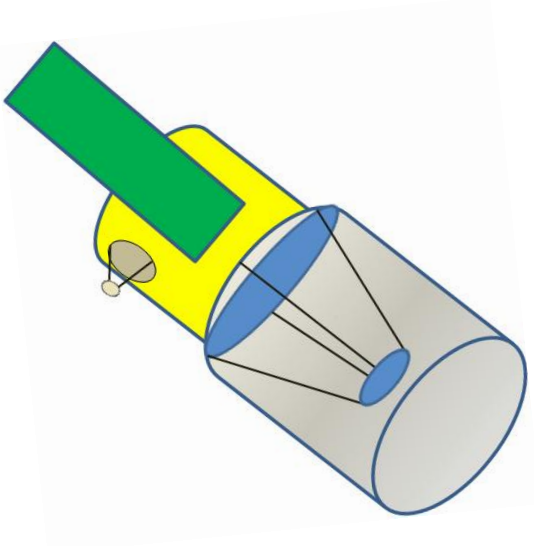


Figure A-1. POTHOLE Spacecraft

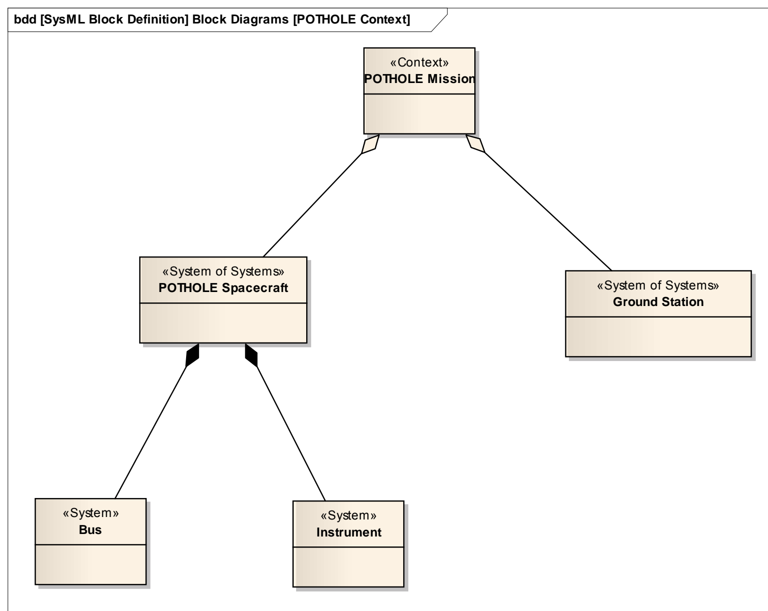
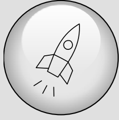


Figure A-2. POTHOLE Mission Block Diagram

We next define a few key business requirements constituting the contract between the customer and the implementer. Business rules define the constraints and functions that must be delivered by the implementer and are the basis for the customer's acceptance criteria.



ROCKET SCIENCE: V&V Challenges

Several challenges are inherent in current V&V methodologies.

- There are multiple layers of requirements (refer to Figure A-3) – mission, systems, subsystems, assemblies, units and components.
- Mission requirements are often driven by capabilities. Instead of asking, “What do you want us to build?” we are often asked, “What do you have and how can we use it?”
- A design is V&V'd through tests, analysis, demonstration and inspection.
- Keeping track of who does what and how lower level V&V rolls up into higher levels is not managed very well.
- Tests at higher levels wait until tests at lower levels have completed. So problems related to validating higher level functions aren’t recognized until later in the development cycle when it is more expensive and time consuming to fix them.
- Models (e.g. aerothermal and reliability models) are often used for various aspects of V&V. These models are not integrated and do not necessarily provide an easy path for incorporating effects in one model into another.

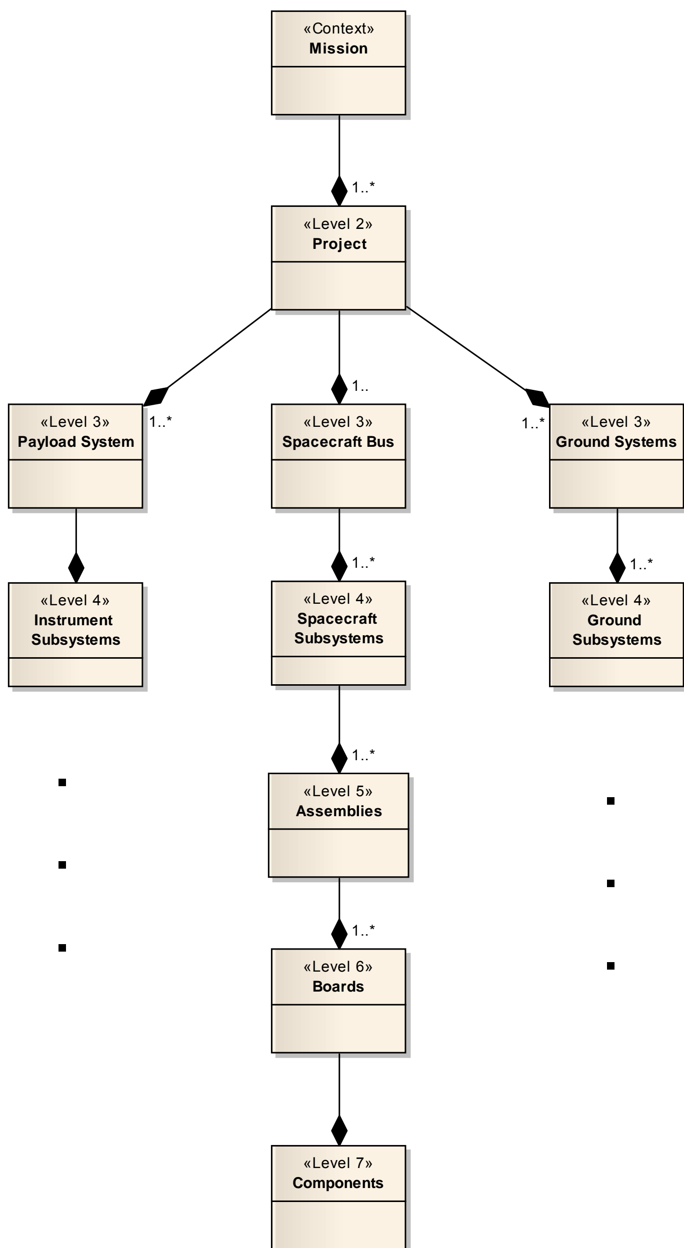


Figure A-3. Typical space system design hierarchy includes up to seven levels of specification and V&V

POTHOLE Business Requirements

Figure A-4 shows the ten requirements, as modeled in Enterprise Architect. Mission requirements derive from the opportunities and limitations in system use, environmental conditions, functional criticality, and the other systems the customer employs in support of the system.

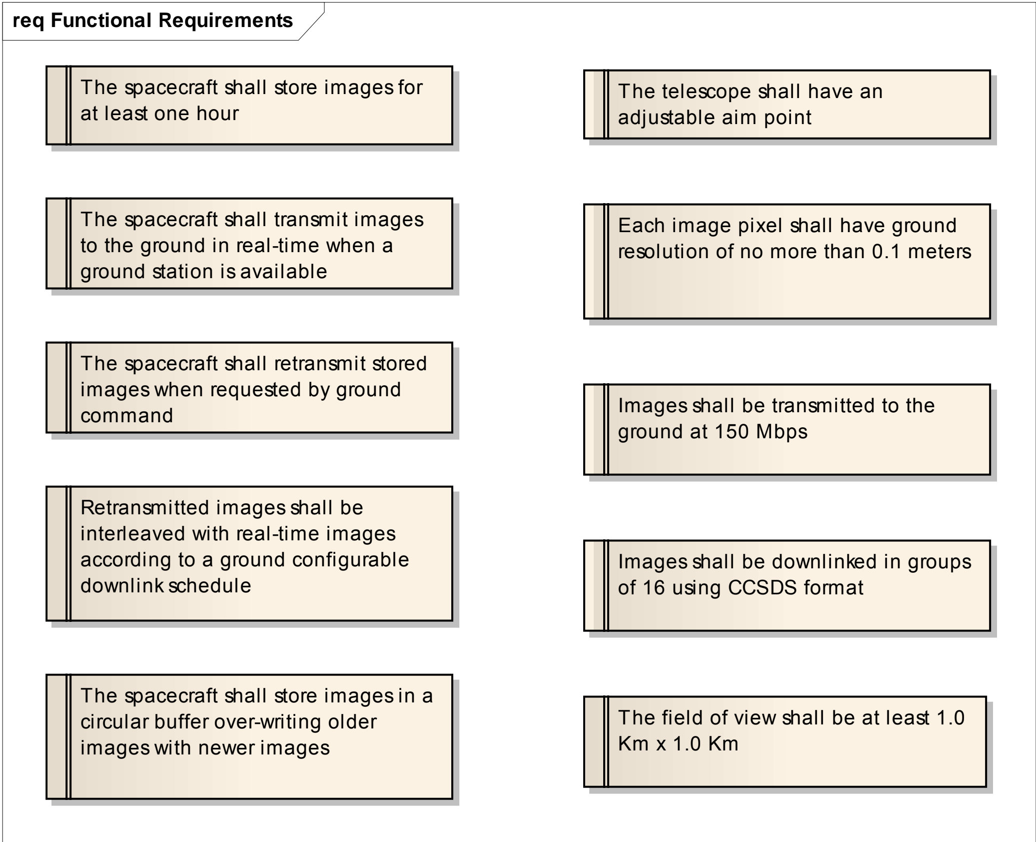
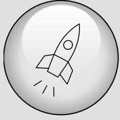


Figure A-4. POTHOLE Mission Requirements



ROCKET SCIENCE: Validation and Verification (V&V)

We define **validation** as the processes performed to assure that we are building the right thing for specified needs. If a user wants to measure magnetic fields around a planet then our design must include a magnetometer. We should not build our spacecraft with something else unless we have discussed it with our customers and they agree that our alternative is acceptable. In a space flight mission, validation focuses on certifying that the system meets a set of mission objectives documented in operations concepts and mission plans while operating in flight-like conditions. Validation is often subjective because it asks whether the system is able to accomplish its mission goals.

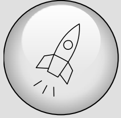
Verification is defined as the processes performed that assure we have built the system right. Verification determines that the system has the right components to do its job and that at each level of design those components satisfy a set of requirements. Collectively, requirements are a formal contract between the customers and implementers of a system. Unlike validation, verification is associated with formal, objective tests that have either a pass or fail result.

Defining comprehensive yet efficient V&V for complex systems is a difficult problem and is often made more difficult by fuzzy requirements, incomplete definition of desired behaviors, multiple teams, and unclear responsibilities. Inefficient V&V resulting in unnecessary overlaps or repeated tests adds costs time and money but is far less dangerous than incomplete or missing tests. Yet testing all combinations and permutations of a complex system is impossible both for the size of the problem and for the lack of visibility into internal operations as systems are integrated.

Typically, a space system is designed top-down and bottom-up. Mission planners think through the type of data they want to collect which drives lower-level allocations and requirements. At the same time, designers are often looking at what hardware and software they can reuse even though they may not yet have a complete understanding of what functions they must provide. Often bottom-up selection and top-down mission analyses and decompositions meet in a integration and test facility where disconnects are frequently found.

The Bus and Instrument structures are further detailed in the block definition diagrams (BDDs) shown in Figures A-5 and A-6 respectively.

There are six major subsystems that form the Bus: Telecom which consists of radios for ground communication. Guidance and Navigation is responsible for know where the vehicle is relative to the ground and for maintaining pointing so that the telescope points to the Earth's surface while the solar array points to the sun. Avionics houses the flight computers on which flight software executes. The Propulsion subsystem includes the thrusters, valves, and propulsion tanks. Temperatures are maintained by the Thermal subsystem and the Power subsystem manages electrical power.



Lunchtime Conversation with a Rocket Scientist

We discussed our fictitious mission requirements with a friendly rocket scientist over lunch. The conversation went something like this:

Well, the first requirement tells us how long we have to keep images onboard. Once we know the image capture rate and image size, we can place a lower bound on how much storage we need. The seventh requirement tells us that our field of view (FOV) is 1.0 Km x 1.0 Km. Assuming we want to take an image in every 1.0 Km x 1.0 Km patch, then we have to take an image every time the ground track of the spacecraft moves 1.0 Km which can be computed from the spacecraft orbit (inclination and altitude).

Pizza arrives…

For example, at an altitude of 20,000 Km (GPS altitude), we need to image roughly every 100 milliseconds at the equator. Requirement 9 tells us that our image sensor is 10K pixels * 10K pixels = 100M pixels. At 12 bits per pixel, each image is 1.2 Gb. Storing images for an hour translates into around 4.3Tb of onboard storage – a large number, but still within the realm of current solid state recorder technology.

Time for a second slice…

In a real design, we would be quite concerned about the practicalities of building a telescope and optics with the required precision. However, for our purposes, we are only interested in data collection, transport, and processing. Our back-of-the-envelope analysis gives us interface rates, storage requirements, image sensor size, and hints of needed processing throughput. We're now ready to define our domain model and use cases.

The Instrument comprises Data Storage for holding images when ground stations are not in view and for retransmission, a Camera that takes images, an Instrument Computer that manages the instrument and image collection functions, and the Telescope which consists of mirrors and optics.

Figure A-7 is an IBD that shows interfaces between the CDH processor, Local Engineering Unit (LEU) used for analog data sampling, Non-Volatile Memory (NVM) used to store critical variables and software images, and the Communications Interface (CommInterface) that sends and receives messages from the telecom radios. We use the highlighted NVM to define interface tests in the validation section below.

«System»
Bus

«Subsystem»
Guidance and Nav

«Subsystem»
Telecom

«Subsystem»
Avionics

«Subsystem»
Prop

«Subsystem»
Thermal

«Subsystem»
Power

«block»
CDH

«block»
FSW

«block»
Processor

«block»
Non-VolatileMemory

«block»
EngineeringDataCollection

«block»
TelecomInterface

Figure A-5. POTHOLE Spacecraft Bus BDD

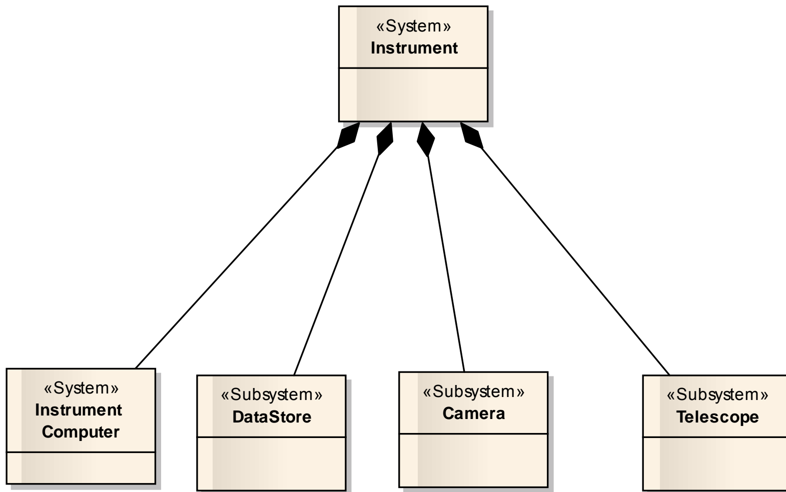


Figure A-6. POTHOLE Instrument BDD

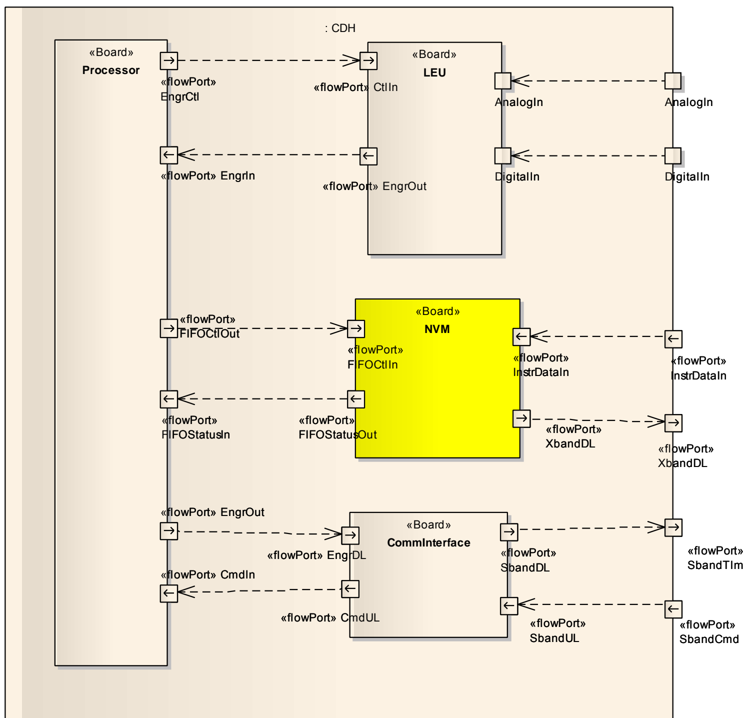


Figure A-7. Examples of CDH interfaces

Earlier we made a case for top-down and integrated V&V, yet here we are showing significant detail for our hypothetical spacecraft. Natural questions might be: how did we come to this partitioning, how do we know it is right, and could there be a better decomposition that does what we need? The simple answer is that these are basic functions of a 3-axis stabilized spacecraft that can capture images and send them to the ground. Within this structure designers are free to allocate specific behaviors and requirements. Moreover, we are free to modify the structure if our V&V process determines that it is not adequate.

Why distinguish between “Bus,” and “Instrument,” couldn’t these be just part of a Flight Vehicle and avoid the intermediate partition? The Bus and Instrument partitions are neither required nor necessary but reflect the reality that different development groups are usually responsible for providing Bus and Instrument functions. These groups independently design and test their portion of the spacecraft and then eventually those systems are brought together and tested. This highlights the point made earlier that traditional V&V waits for verification of lower level functions before testing higher level functions. Electrical and functional interface documents define how this should happen and when everything is properly specified and implemented then all works out. But, that’s not always the case.

POTHOLE Domain and Use Case Models

We have a good idea of what behaviors are needed in the POTHOLE spacecraft for collecting and transmitting images to the ground for pothole analysis. Those behaviors also help define our domain model. Domain models and use cases are tightly coupled because use cases refer to domain elements and domain elements must be present in at least one use case. Consequently, domain and use case model interactions are the basis of initial model validation because each model checks the other. Neither model can be considered complete or correct unless each is consistent with and supportive of the other.

From the requirements and mission goal, we know that our domain model must have an "image" element. We also know that individual images are packaged on the spacecraft prior to downlink, so our domain needs a concept of "image collection" which is composed of "images." Similarly, we know that we will need some sort of "image queue" in our domain model that feeds images to analysis software because images cannot be processed concurrently.

The domain model in Figure A-8 and the corresponding use case model in Figure A-9 result from iterating domain elements and use case descriptions. Figure A-8 populates model elements with conceptual function definitions. These may need revision as more detail is known, but at this point, the elements and functions are consistent with the use cases in Figure A-9 and the mission requirements.

The domain model includes the Instrument that produces an Image Collection which is created as an aggregation of individual Images. The Image Collection has an association with the Ground Station through the Telecom subsystem. The Ground Station maintains an Image Queue for analyzing and annotating the Images. Images annotated with Potholes are sent to the Subscriber by the ground station.

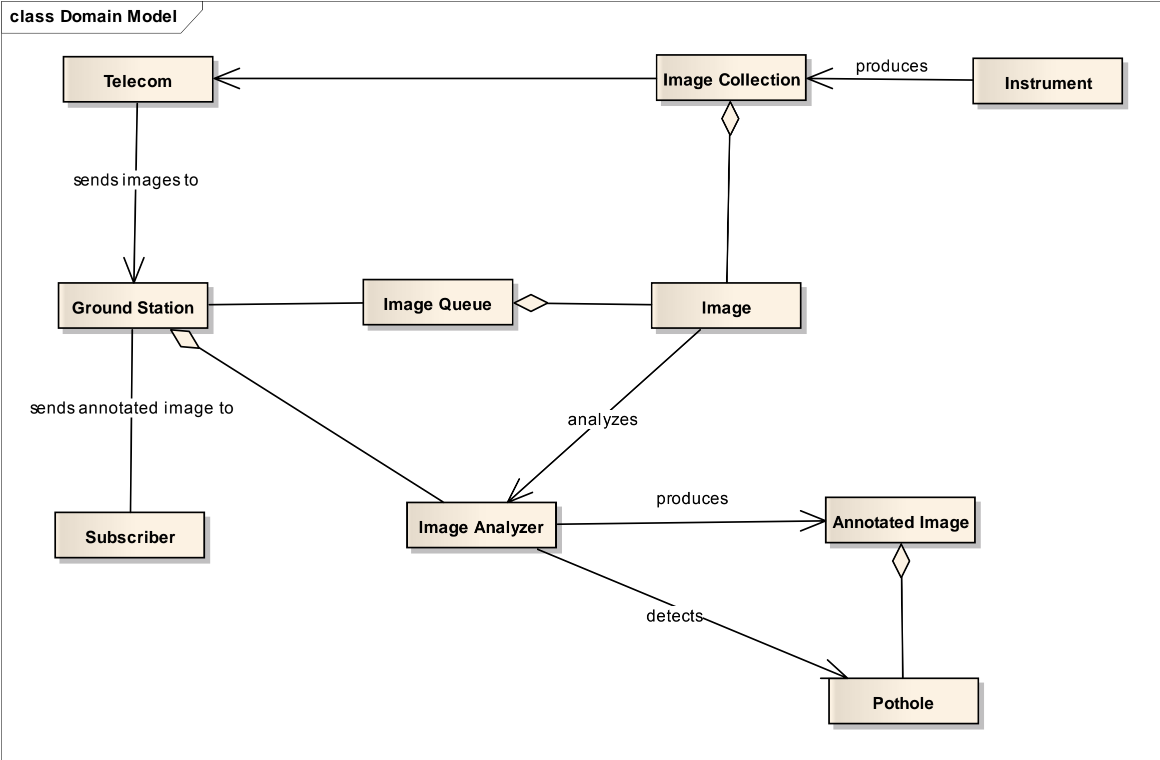


Figure A-8. POTHOLE Imaging Domain Diagram

The mission-level use cases for our reference function shown in Figure A-9 complement the domain model and reflect our understanding of the imaging collection and analysis process. The spacecraft must achieve and maintain the correct orbit (Maintain Orbit). The instrument must be aimed at a point on the ground point prior to looking for potholes (Aim Instrument). Spacecraft health must be maintained throughout the mission (Maintain Spacecraft Health) including dealing with spacecraft emergencies (entering safemode which keeps the spacecraft power positive and in communication with the ground). When commanded, we capture and process images (Look for Potholes) and if images are corrupted in transmission, the ground may request that they be retransmitted (Retransmit Images).

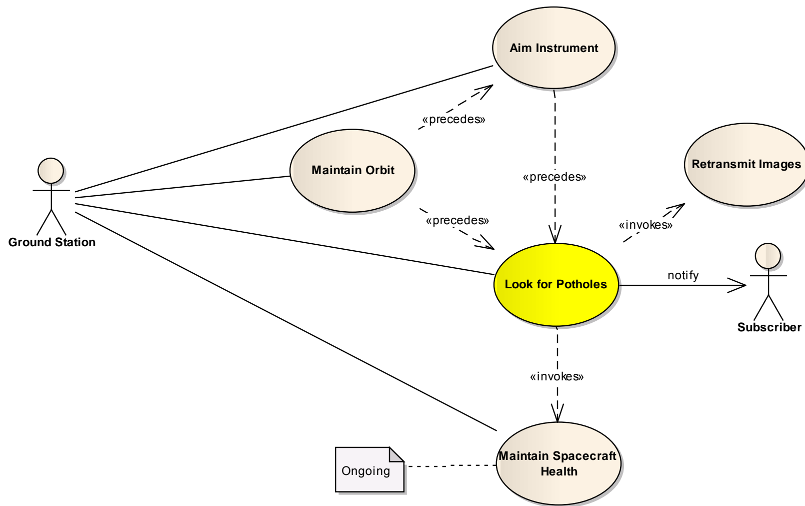


Figure A-9. POTHOLE Mission-Level Use Cases

Because our primary mission is pothole detection, we focus on the "Look for Potholes" scenario that collects images with sufficient resolution to detect large potholes and sends the images to the ground for processing. More precisely:

Basic Course:

Ground Station sends a command to **Spacecraft** to collect **Images**.
Instrument collects **Images**, resulting in an **Image Collection**
Telecom sends the **Image Collection** to the **Ground Station**
Ground Station adds **Images** to **Image Queue**
Image Analyzer gets **Image** from **Image Queue**
Image Analyzer detects **Potholes** and produces **Annotated Image** showing **Pothole** locations
Image Analyzer sends **Annotated Image** to **Subscriber**

Alternate Courses:

Ground Station sends a command to **Spacecraft** to retransmit **Image Collection**: Invoke ***Retransmit Images***
Spacecraft enters safemode: invoke ***Maintain Spacecraft Health***
Ground Station sends a command to **Spacecraft** to stop collecting images: invoke ***Maintain Spacecraft Health***

A few things are worth pointing out in this "ICONIX process style" use case[55]. The scenario is written precisely using nouns (shown in bold) that represent objects in the domain or context models. Verbs apply actions that transform one noun into another noun or transport a noun to another action. Bold italicized text refers to use cases in the use case diagram.

Comparing the nouns and verbs in our use case with the domain model elements gives us confidence that our models are consistent. But we don't yet know that we defined the use case scenario correctly and so we can't yet claim that we have validated our initial concepts. Our next step creates a conceptual design in the form of a robustness diagram[56] that facilitates a design walk-through for validating the use cases. We will see later that the robustness diagram is also an essential tool for defining test cases.

POTHOLE Conceptual Design

Figure A-10 shows a robustness diagram for the Look for Potholes use case. In a robustness diagram, the logical and conceptual structures are analyzed ensuring that the use cases it models are sufficiently robust for the intended usage behaviors and requirements. Robustness diagrams are annotated with requirements establishing completeness and consistency design. Missing, incorrect, and incomplete requirements become readily apparent.

A robustness diagram comprises: actors (people or external systems that interact with the system under consideration, boundaries (interfaces that actors interact with), controllers (the verbs in the use case statements), and entities (elements from the domain or context models).

A few simple interaction rules define the construction of robustness diagrams:

- Entities and controllers may interact.
- Actors and boundaries may interact.
- Boundaries and controllers may interact.
- Entities cannot directly interact with other entities.

In Figure A-10, circles with arrows represent controllers, underlined circles are entities and circles attached to vertical lines are boundaries. Requirements are associated with controllers. Every controller must have at least one requirement so Figure A-10 is not complete. Moreover,

---

[55] *Use Case Driven Object Modeling with UML: Theory and Practice* by Doug Rosenberg and Matt Stephens.

[56] Ivar Jacobson, et. al. (1992), *Object Oriented Software Engineering - A Use Case Driven Approach*, Addison Wesley.

every requirement must be associated with a controller and missing associations mean either omissions in the robustness diagram or unnecessary requirements.

We can walk through the *Look for Potholes* steps and follow the flow in the robustness diagram. Inconsistencies are reconciled by altering robustness elements to reflect the use cases it represents and/or changing the use cases to mirror the robustness diagram. Since updates to use cases may also cause domain model revisions, when iterations are complete, the robustness, use case, and domain models are self-consistent.

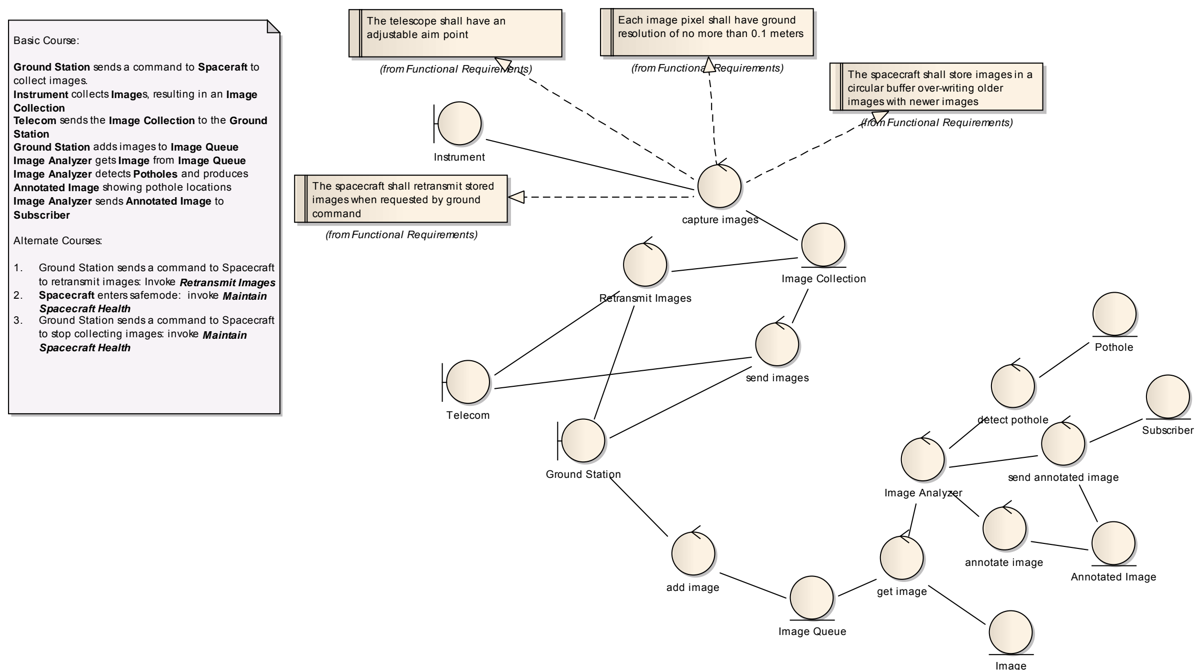


Figure A-10. POTHOLE Robustness Diagram with a few example requirements

Figure A-11 shows a portion of the sequence diagram associated with the Look for Potholes use case. Behaviors are allocated to the system elements defined in the domain model and block diagrams. Each behavior must be unit tested, as shown in Figure A-25.

The Extended Domain Model in Figure A-12 shows an object-oriented allocation of behavior to system elements. Allocations satisfy all behaviors specified in the use case as shown in the sequence diagram.

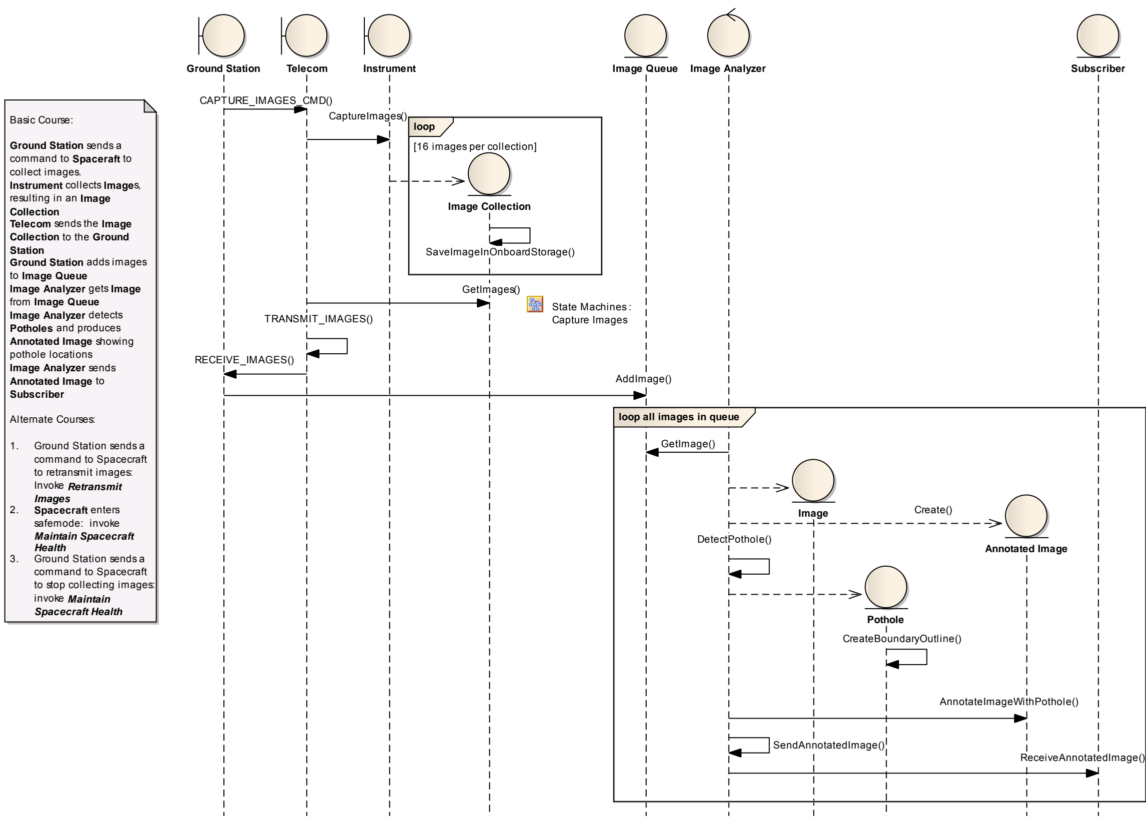


Figure A-11. Sequence diagram shows what the system must do and which elements are responsible for each behavior

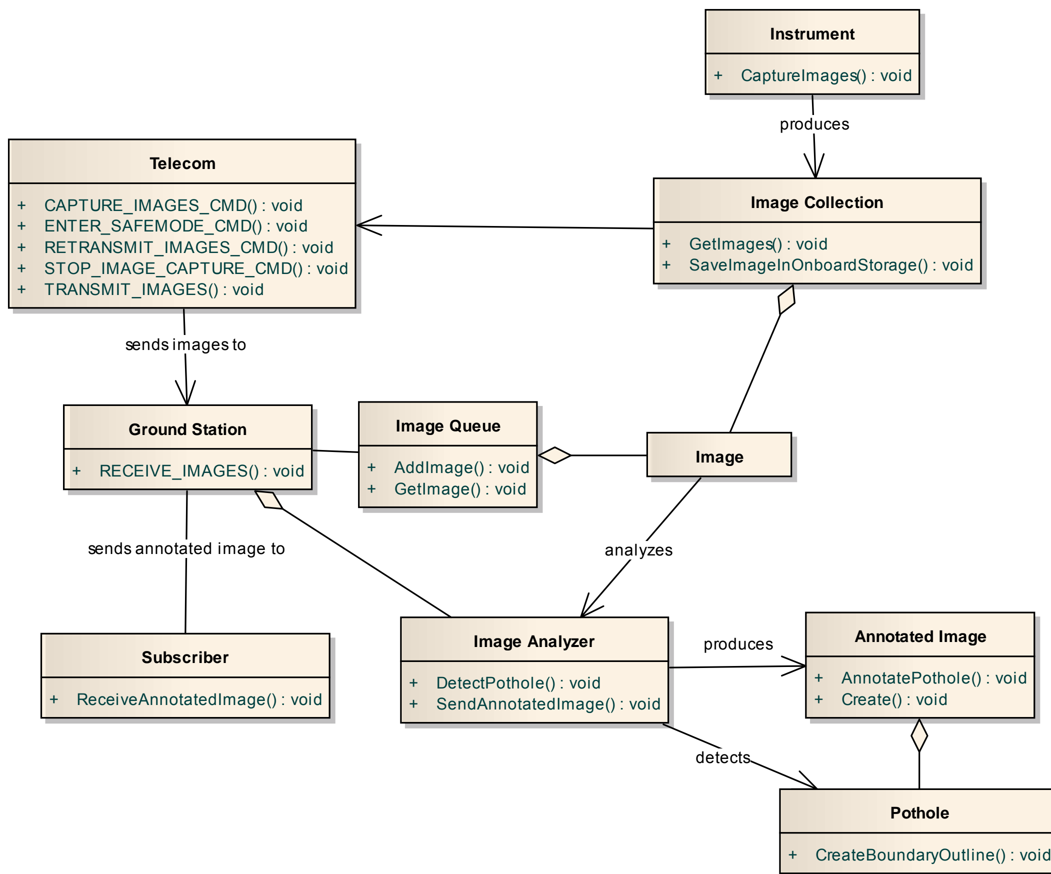


Figure A-12. Expanded Domain Model showing behaviors allocated to system elements

Figure A-13 shows the state diagram for initializing the instrument and capturing images. After initializing buffers and the camera, the instrument enters the Idle state where it waits for a ground command. Receiving a CAPTURE_IMG_CMD causes a transition to the Capturing Images State in which the camera is made ready.

Images are captured and saved by the “do” behaviors. Images are captured in the circular buffer until the STOP_IMAGE_CAPTURE_CMD is received. This command disables the camera after the current image capture is complete and the instrument returns to the Idle state.

Figure A-14 shows a power parametric model comprising camera, data store, telescope, and instrument computer power models. The parametric model sums these components to form a total power usage.

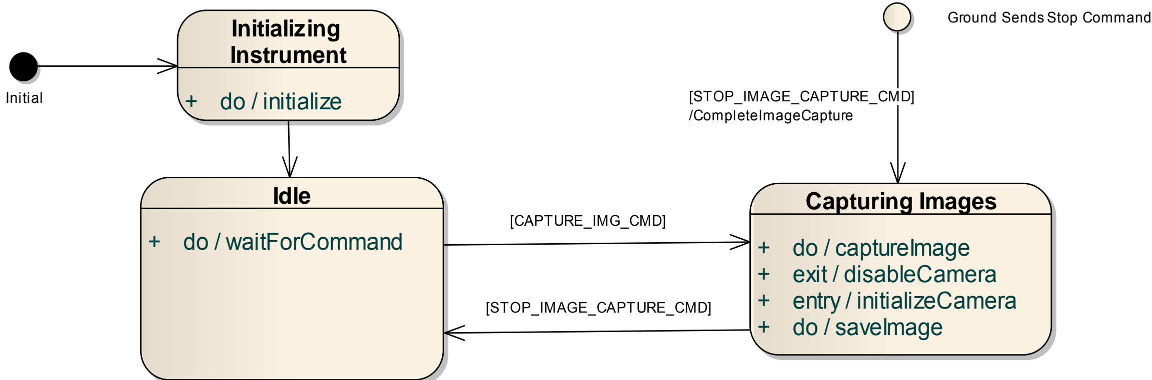


Figure A-13. All states and transitions must be tested

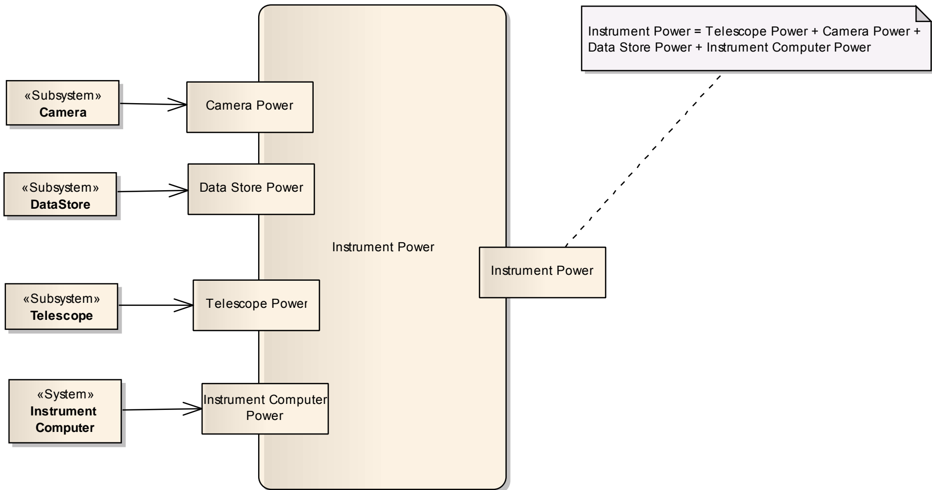


Figure A-14. Instrument Power Parametric Model

This completes our description of basic SysML artifacts and sets up the testing discussion that follows. While our POTHOLE example is simplistic and fanciful, the models we described are the basis for many systems.

The next section describes the baseline DDT methodology and rationale. We follow with a description of an extension for systems.

Design Driven Testing

Stephens and Rosenberg[57] define Design Driven Testing (DDT) as a logical extension to UML-based software design. The premise of DDT is that the bottom-up "agile" practice of driving designs from unit tests (test driven design – TDD) is inherently backwards. In TDD, test cases are developed before code design. Code is then written to make the test cases pass and recoded as necessary if test cases fail. TDD produces a lot of test cases but more seriously because of its ad hoc nature, doesn't fully account for a customer's needs. In fact, the user needs are determined by the test results!

Arguably, ignoring the cost of unnecessary testing and recoding, TDD works in an agile development cycle. However, the type of code produced is not suitable for critical applications and most users can tolerate odd behavior until the next update is pushed out. No one would ever develop spacecraft or any other critical applications using TDD.

The DDT approach follows a top-down methodology in which the testing is automatically generated from software models. Automation assures that test cases aren't missed and the link to software models assures that test cases are driven from the design.

DDT is an appropriate starting place for developing Systems V&V test methods because most system behavior is accomplished in software. We define SysML based extensions to DDT (DDT/Systems) later in the paper. Since we are starting from DDT, we present a high level summary of the approach here, which we will illustrate by example shortly.

DDT automates test case generation for both developers using unit tests and for QA testers using acceptance tests. Figure A-15 shows acceptance test relationships. Mission requirements are driven by customer needs and use case scenarios define what a user does with the system. DDT generates tests that evaluate requirement compliance and correct implementation of desired behavior. As shown in Figure A-16, DDT also applies during the development cycle by automating creation of controller tests from robustness models (see next section) and unit tests.

---

[57] *Design Driven Testing: Test Smarter, Not Harder* by Matt Stephens and Doug Rosenberg (Apress, 2010). Also see: www.designdriventesting.com

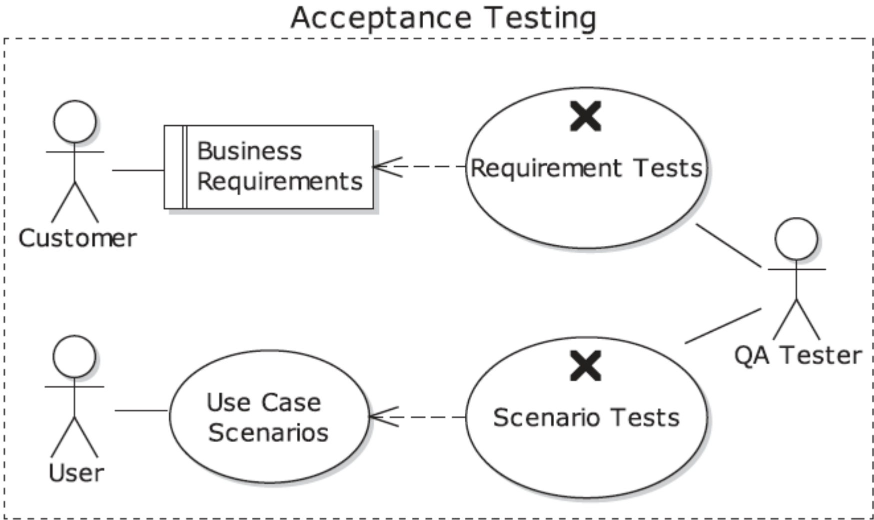


Figure A-15. Acceptance Testing Flow

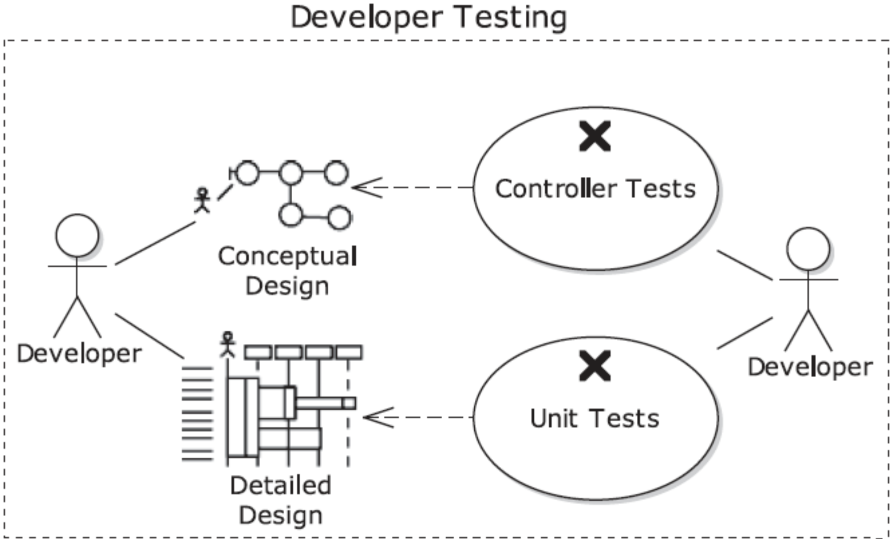


Figure A-16. Developer Testing Flow

The DDT methodology can be summarized as:

- Generate test cases from requirements.
- Generate thread tests from scenarios.
- Generate unit tests from conceptual and detailed designs.

We will illustrate Requirement, Scenario, and Unit test generation by example in a few pages. However, these tests are not sufficient to test all aspects of a hardware/software system. Our next section describes an extension to DDT suitable for embedded systems.

DDT/Systems Extensions

Extending DDT for systems necessitates covering additional elements as shown in Figure A-17.

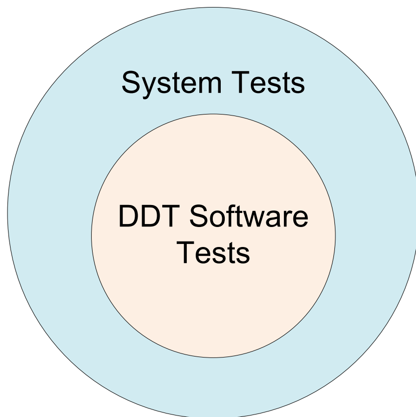


Figure A-17: DDT/Systems includes DDT Software and additional tests

These elements include:

- State Machines
- Activity Diagrams
- Block Definition Diagrams
- Internal Block Diagrams
- Constraint Block Diagrams

Figure A-18 shows the system extensions to DDT.

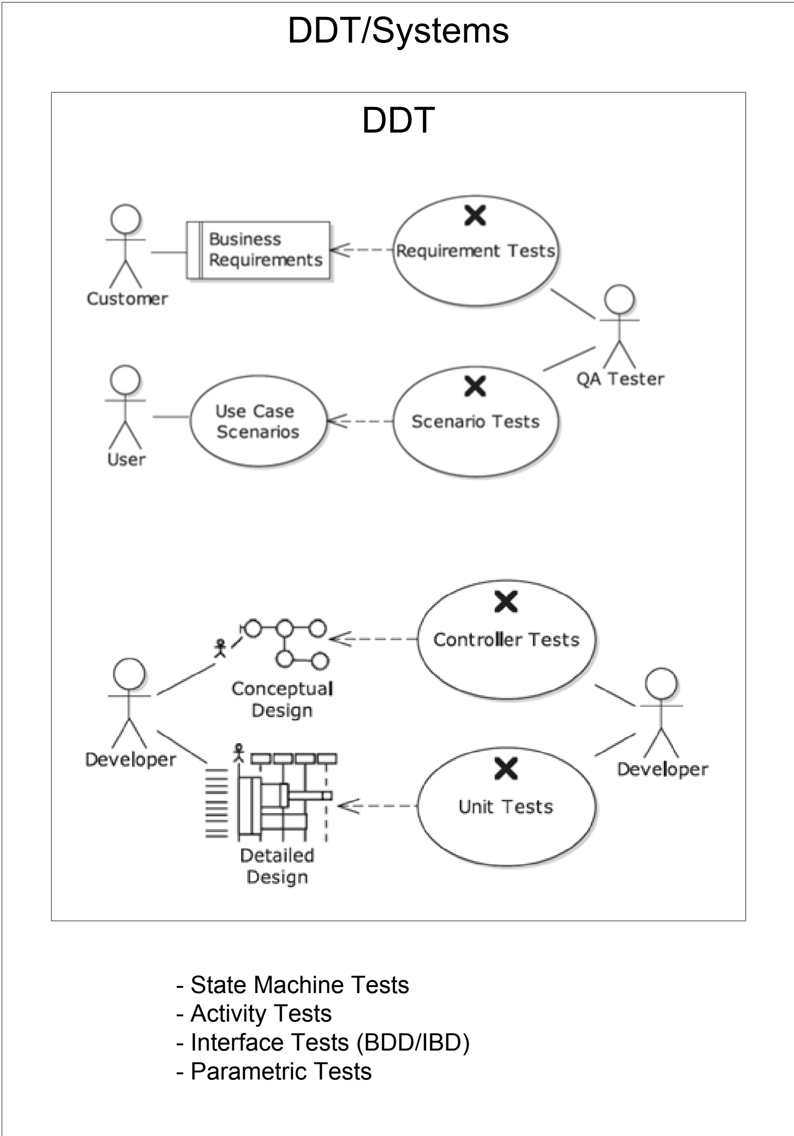


Figure A-18: DDT Systems extends DDT for software with additional system tests

We next apply DDT/Systems to the POTHOLE spacecraft.

Validating POTHOLE with DDT/Systems

We focus our POTHOLE design and V&V examples on a single reference function: taking and processing images for finding potholes. We walk through a simplified V&V process by example in the figures below. The next section summarizes the real-world V&V process.

Now we come to the next validation step: checking the completeness, correctness, and consistency of our use case description. We want to assure ourselves that we haven't forgotten something or specified something too vaguely or incorrectly. Any errors in our use case must be fixed and as necessary and the domain model iterated.

We are now ready to define test cases. We can automatically generate a test case element for every requirement on a requirement diagram, as shown in Figure A-19. Requirement tests are an integral part of the test plan. Generating test cases automatically from all requirements ensures that none are forgotten.

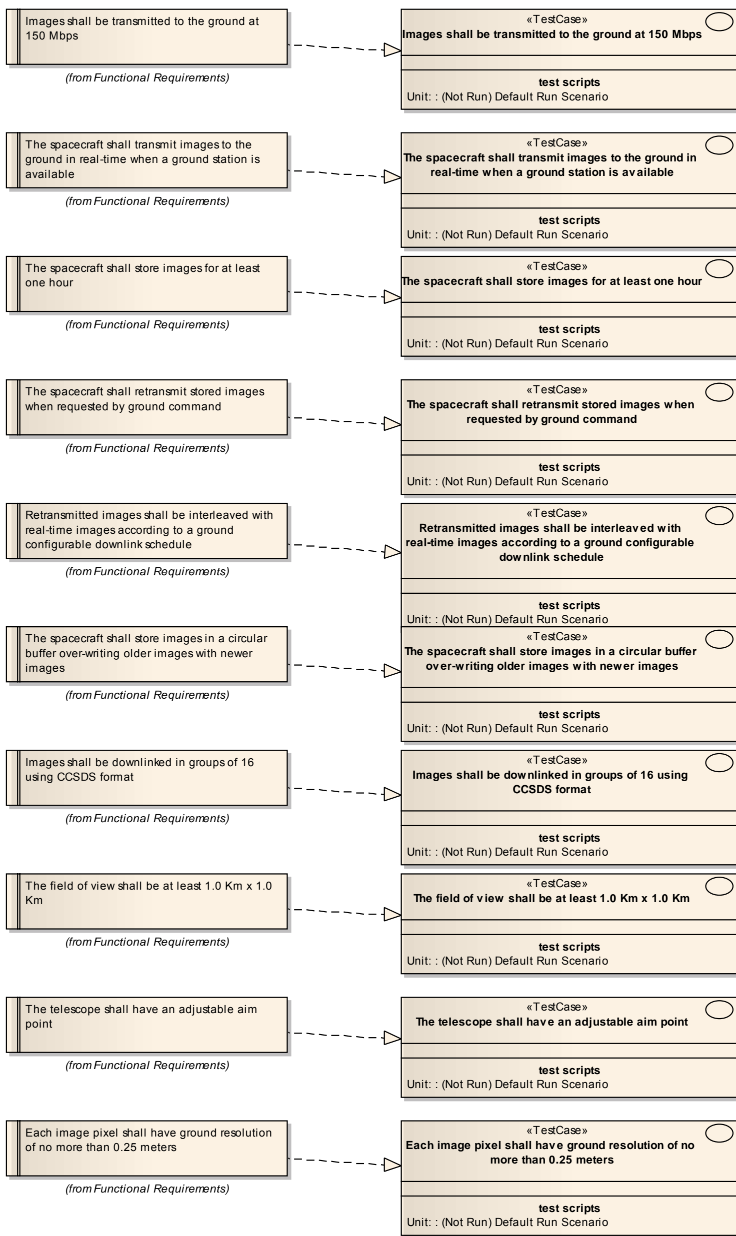


Figure A-19. POTHOLE Mission Requirement Test Cases

We use a “use case thread expander” to automatically generate test scripts for all sunny-day/rainy-day permutations within a use case. The premise of this approach is that a use case with one sunny-day scenario and three rainy-day scenarios requires at least four test scripts in

order to exercise all usage paths. This process is illustrated in Figures A-19 to A-22. Figure A-20 shows a “structured scenario” that specifies step numbers and “join” logic for our use case.

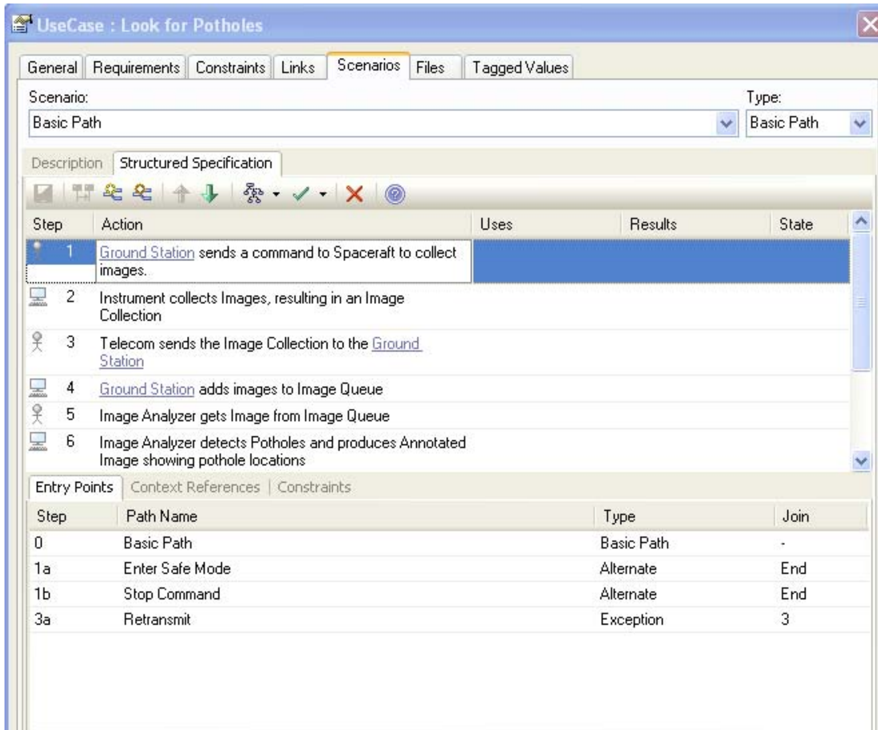


Figure A-20. Expanded sunny/rainy day threads: create a structured scenario

Figure A-21 shows an automatically generated activity diagram that is used to verify the logic of the scenario before tests are generated.

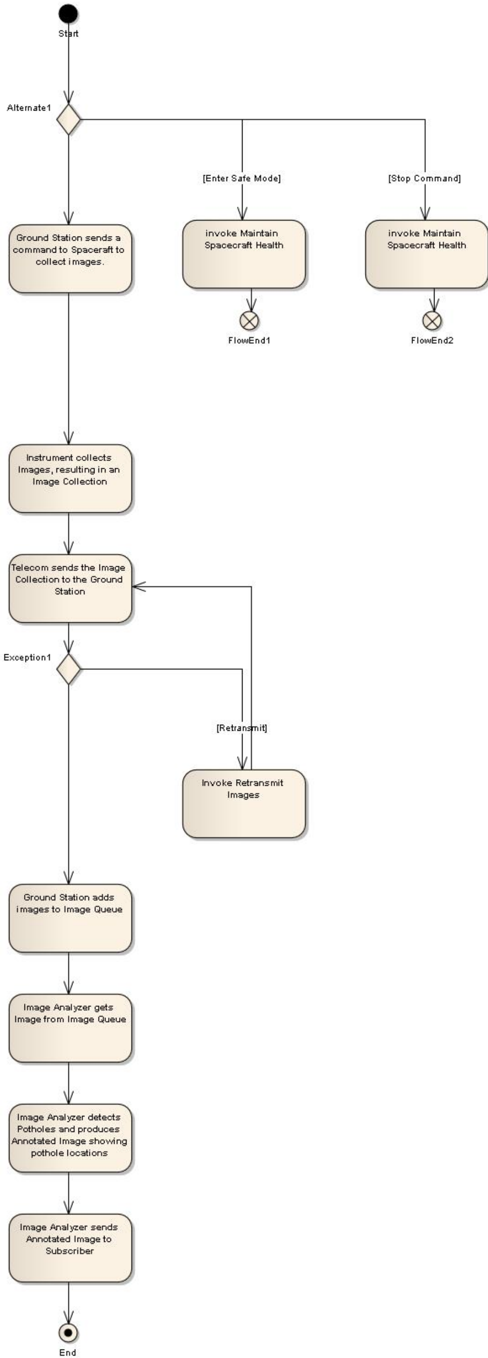


Figure A-21. Generate an activity diagram and check flow against use cases

Figure A-22 shows a test case diagram, showing the complete use case text in a note, and a test case element with a test scenario ("thread test") for each Course of Action within the use case.

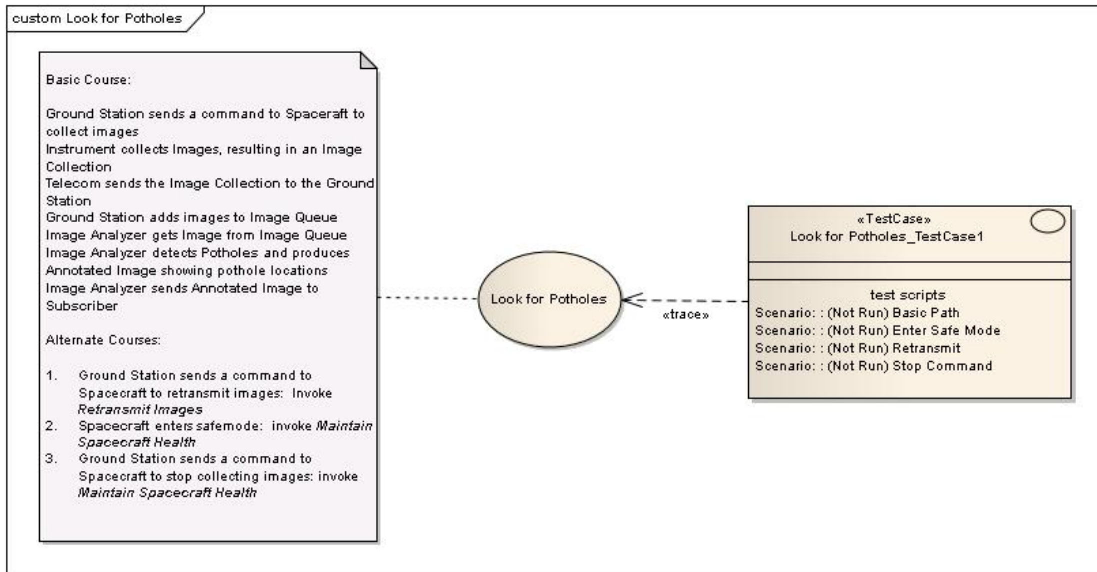


Figure A-22. Scenario Testing

Usage paths are automatically generated for each thread, as shown in Figure A-23. These threads show portions of the Basic Course and portions of the Alternate Courses, as relevant for each thread.

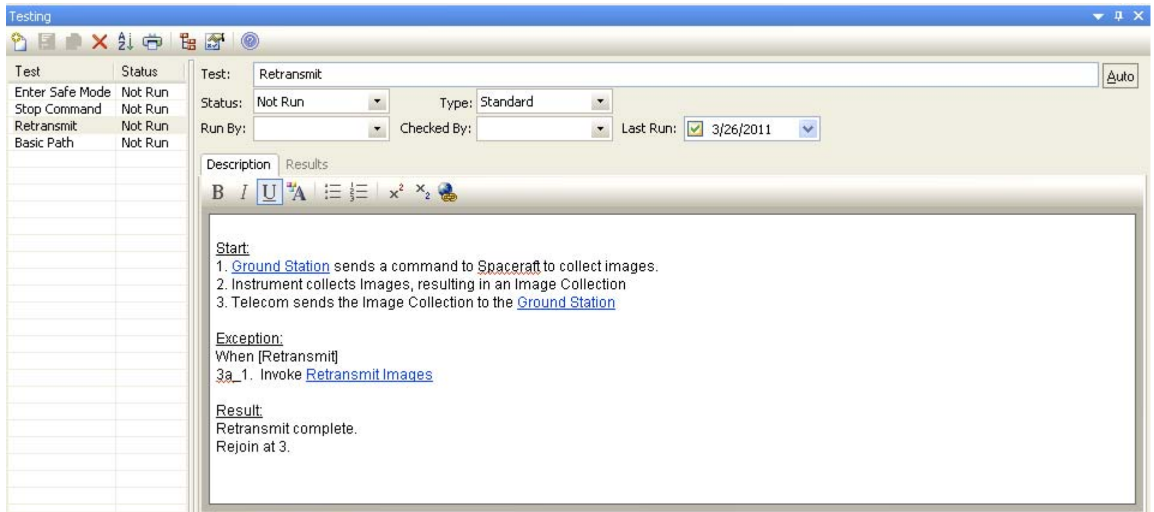


Figure A-23. Generate test scripts from use case text

DDT follows a use case driven approach to design in which each use case performed by the system is first elaborated at a conceptual level (refines the use case and validates the system) and then at a detailed design level (for system verification). DDT generates test case elements from “controllers” on conceptual design (robustness) diagrams, as shown in Figure A-24.

Unit tests are also generated from messages on design-level sequence diagrams as shown in Figure A-25. After test scenarios have been detailed with Inputs, Outputs, and Success Criteria, the test cases are automatically transformed into unit test code suitable for regression testing frameworks such as JUnit, NUnit and FlexUnit. [58]

[58] See: www.junit.org, www.nunit.org and www.flexunit.org respectively.

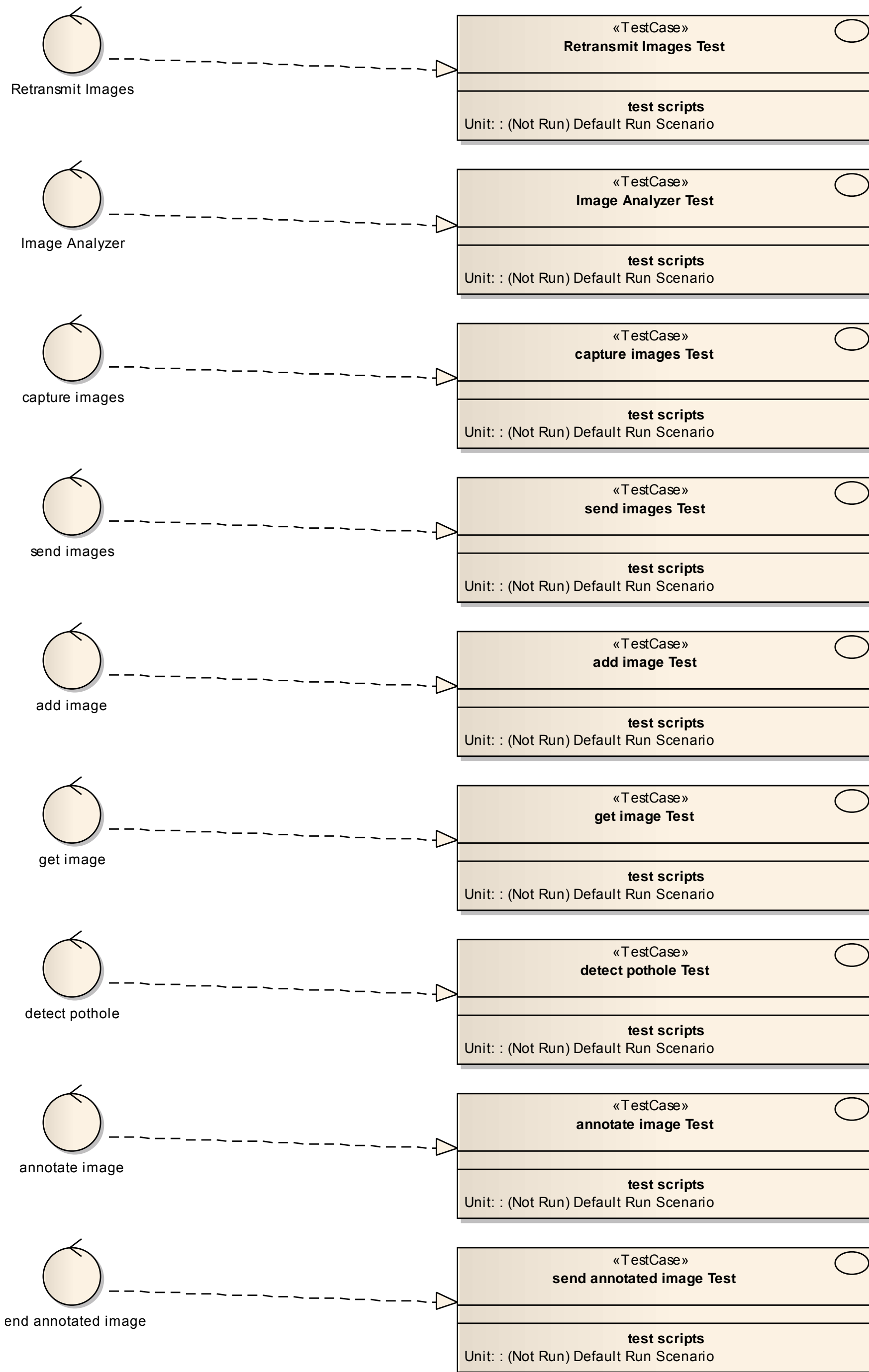


Figure A-24. POTHOLE Controller Test Cases

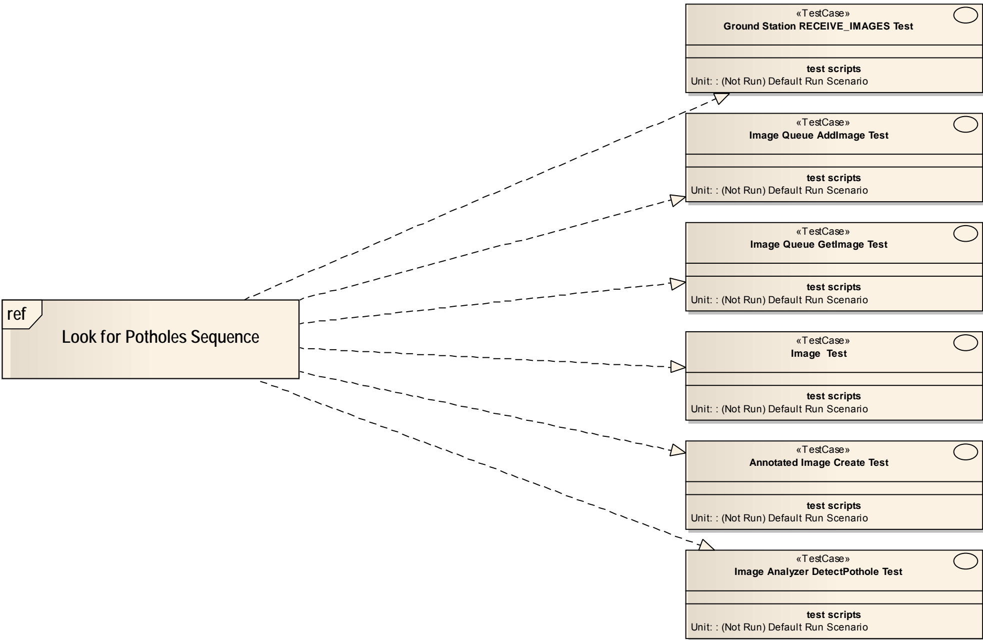


Figure A-25. Subset of Autogenerated Look for Potholes Sequence Message Tests

Figure A-26 shows state, state transitions, and behavior test cases for the capture image state machine described earlier. All behaviors, transitions and triggers must be tested including "entry," "do," and "exit" behaviors on states. Note that similar tests must also be defined for activity diagram behavioral descriptions.

Interface Control Definitions must also be tested. This includes electrical and logical interface definitions as shown in Figure A-27. This includes port, required/provided interfaces, electrical and timing, and fault behavior tests.

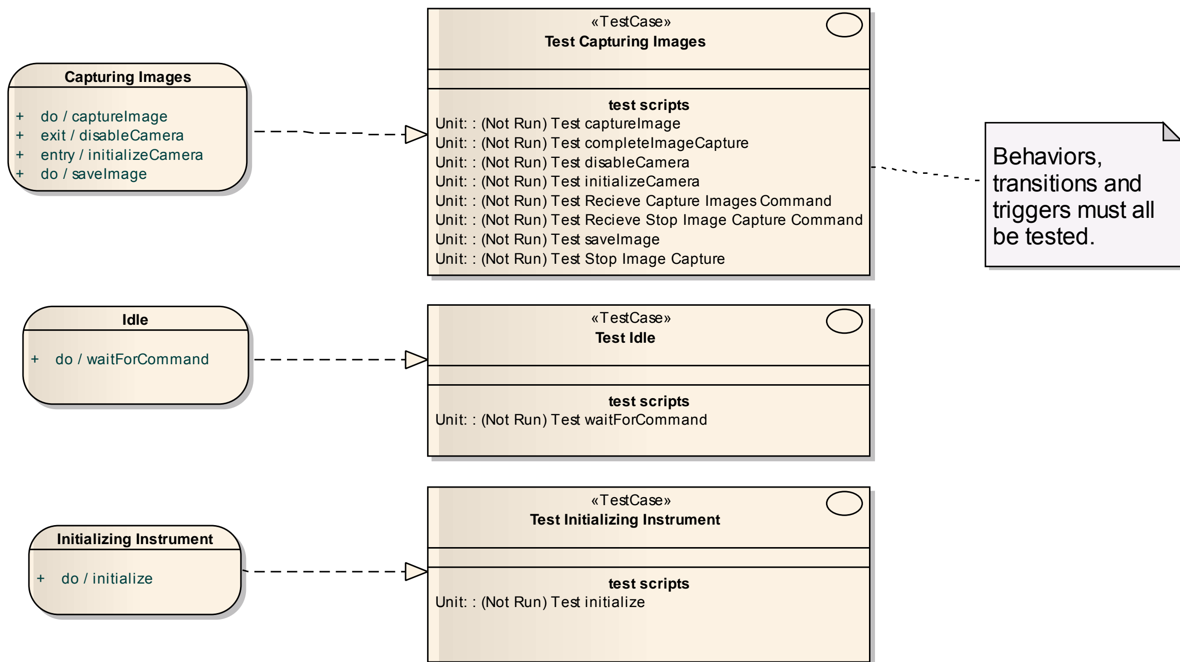


Figure A-26: Examples of state tests

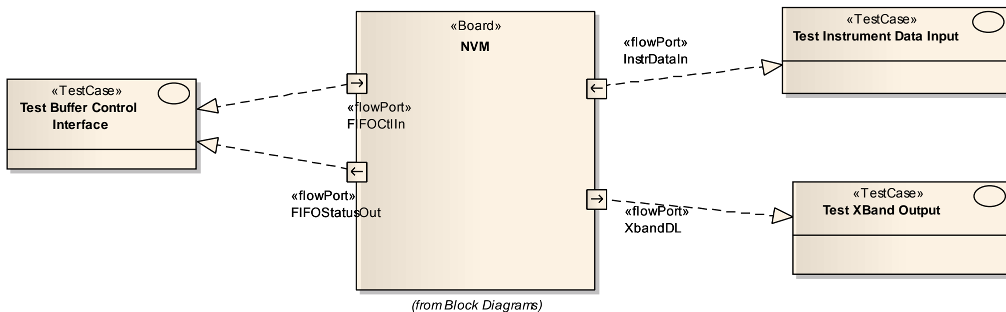


Figure A-27. Need to test all behaviors and interfaces defined in an Interface Control Document (ICD)

In many cases, analytical models are needed for evaluating non-behavioral requirements. These models could evaluate mass, power, radiation effects, and so forth. Figure A-28 shows a parametric diagram that evaluates instrument power. The computed total power is compared to a maximum instrument power requirement.

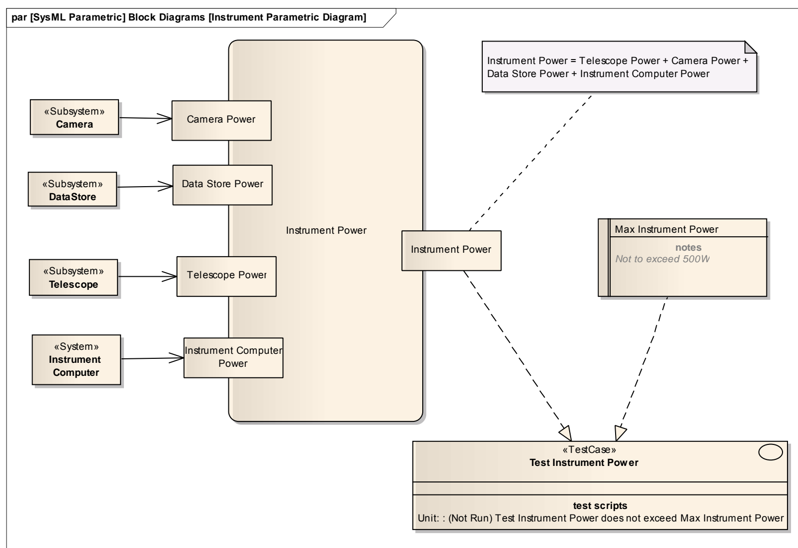


Figure A-28. Parametric evaluation of instrument power

This concludes our illustration of DDT/Systems using the POTHOLE example.

As illustrated in this section, DDT/Systems comprises the following set of tests:

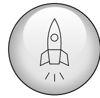
- Requirement tests
- Scenario tests
- Controller tests
- Sequence tests
- State machine & Activity tests
- Interface tests
- Parametric tests

This is the most comprehensive and efficient system V&V approach we are aware of.

Conclusions

- DDT automates generation of test cases from UML models for software
- Systems V&V requires additional modeling and test case generation from SysML
- Our fictitious POTHOLE spacecraft illustrates some of these extensions
- We have described proposed addition to DDT that encompasses systems (DDT/Systems)

COMING SOON FROM



fingerpress

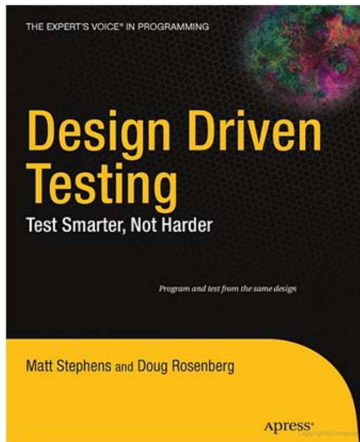
HAMSTER DRIVEN DEVELOPMENT

(it's not for the squeamish…)

Satire from the coding-room floor – from the authors of *Design Driven Testing: Test Smarter, Not Harder* and *Extreme Programming Refactored: The Case Against XP* (featuring *Songs of the Extremos*)

www.fingerpress.co.uk

DESIGN DRIVEN TESTING



The groundbreaking book *Design Driven Testing* brings sanity back to the software development process by flipping around the concept of Test Driven Development (TDD) – restoring the concept of using testing to verify a design instead of pretending that unit tests are a replacement for design.

Anyone who feels that TDD is "Too Damn Difficult" will appreciate this book.

Visit the website for more info:
www.designdriventesting.com

Design Driven Testing shows that, by combining a forward-thinking development process with cutting-edge automation, testing can be a finely targeted, business-driven, rewarding effort. In other words, you'll learn how to test smarter, not harder.

OPEN ENROLLMENT TRAINING

You've read the book, now get the training!

Learn how to apply the process roadmaps you've just read about in this book from Doug at an ICONIX Open Enrollment workshop

Developers: Learn a rigorous, systematic path from requirements to source code
Business Analysts: Accomplish business process modeling for requirements elicitation and process reengineering
Systems Engineers: Apply a rigorous design and testing methodology using SysML

The Design Driven Testing course uses as its example an interactive hotel mapping system which you can try out for real at: www.vresorts.com

For the latest Open Enrollment schedule, please visit:
www.iconixsw.com/EA/PublicClasses.html