METHODS & TOOLS

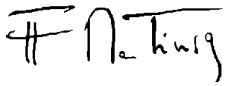
Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Fall 2009 (Volume 17 - number 3) www.methodsandtools.com

The Eleventh Commandment of Software Development

During my software development career, I have seen many software development approaches or methodologies used in different organizations. When a new and supposedly better approach is promoted, some people tend to adopt it with an attitude that could sometimes be close to a religious zeal. It is somewhat funny to see software companies now employing "product evangelists". As a consequence, new believers are often rejecting completely what has been done previously and adopting a "us against them" attitude, especially when the new approach is still a small movement fighting against an established methodology.

This has been the same for Agile, the latest approach on the block, even if the words of the agile manifesto ("while there is value in the items on the right, we value the items on the left more.") try to avoid this "reject the past" situation. Most agile coaches agree also that it is more important to make choices according to the context than to apply a prescriptive checklist, but for some people it is easier to follow blindly a model than to absorb a philosophy. Most of you know the proverb: "Give a man a fish, and he'll eat for a day. Teach him how to fish and he'll eat forever". It is however easier to learn how to work with JUnit than to create efficient unit testing cases. Pure agilists quickly fond their "enemy": the Waterfall, a dinosaur of software process that should be lead to extinction, buried with fossilized Cobol programs in the mainframic stratum of the history of software development. In this issue, we host an interesting article from an organization that mixes CMMI and Scrum. Is this an heretic position for adopters of both approaches? Maybe. Do this article tells us that everybody has to do the same things? No. It is just what the company thought is the best for their software development projects in their own context. Does it seem to be a successful approach for them? Yes. So, this is my eleventh commandment of software development: forget the first ten! If you could summarize software development definitively in a small number of rules, Methods & Tools would not exist. You should read everything you can about every approach with a critical but open mind and choose the good tools for your specific context. Nobody detains THE truth valid for all software development projects. We all make mistakes, which is good because you can learn from them.



Inside

Mature Scrum at Systematic

Carsten Ruseng Jakobsen crj@systematic.dk, Jeff Sutherland, jeff@scruminc.com
Systematic Software Engineering, Scrum Training Institute

Abstract

Systematic is a CMMI level 5 company, where the adoption of Lean led to a unique way of working based on synergies between CMMI, Lean, Scrum and other agile practices.

Lean provides the principles and values needed for high performance teams and organizations, as demonstrated by Toyota for more than 50 years. CMMI provides the process descriptions and support for what disciplines to consider in order to be successful. Agile approaches like Scrum provides best practices and methods for working according to the values of Lean and adopting change as it occurs. Lean, CMMI and Scrum are strong by themselves, but they can be combined to amplify each other – and that is what Systematic has done.

The experiences from combining Lean, CMMI and Scrum have led Systematic to identify examples of explicit guidance from CMMI that help to execute Scrum activities even better. These activities can be implemented based on Lean values and principles and by doing so Scrum can be augmented and matured to ensure that even larger and more complex projects in the future can and will benefit from Scrum.

1. Introduction

Systematic was established in 1985 and employs more than 500 people worldwide with offices in Denmark, Finland, USA and the UK. It is an independent software and systems company focusing on complex and critical IT solutions within information and communication systems. Often these systems are mission critical with high demands on reliability, safety, accuracy and usability.

Customers are typically professional IT departments in public institutions and large companies with longstanding experience in acquiring complex software and systems. Solutions developed by Systematic are used by tens of thousands of people in the defense, healthcare, manufacturing, and service industries

This paper presents how Systematic combined Lean, CMMI and agile during the past 5 years. What were the different steps and what are the experiences and lessons learned. Agile purists and small agile projects may find our experiences and suggestions non-agile or counter-productive and very large high maturity organizations may find our experiences too undisciplined.

The experiences are fit for Systematic, but may not be fit for your project or organization. Some companies may be in a similar context to Systematic, and find our experiences directly applicable to their challenges. Others may be in a different context and may find that the answers represented by these experiences are not applicable, and instead find inspiration in the questions asked leading to Systematic's experiences.

2. Why CMMI, Lean and Scrum

Experience. Lean established a new mindset leading to identification of significant improvements

Experience. Scrum implements many of the tools and principles in Lean Software Development

Experience. Scrum implements a subset of project management when interpreting CMMI in a Lean context

Systematic was appraised 11 November 2005 and 15 May 2009 using the SCAMPI (Capability Maturity Model Integration, and SCAMPI are service marks of Carnegie Mellon University)method and found to be CMMI level 5 compliant. Implementing CMMI was the main driver for improvements until November 2005 and therefore as part of yearly strategic planning in the summer 2005, Systematic discussed how to ensure a continued flow of significant improvements after achieving CMMI level 5. The result of the discussion was that future improvements are expected to be implemented applying the CMMI capability established, but using primarily a Lean perspective on new improvements, may help identify new substantial improvements that may be overlooked in a focus on processes alone.

Lean has a proven track record for more than 50 years at Toyota and has been applied successfully to many other industries, and Mary Poppendieck has with her book "Lean Software Development" provided a solid basis for understanding how Lean is applied to software development, described through 7 principles and 22 thinking tools, [1].

In order to understand how to apply Lean Software Development, Systematic analyzed the causality of the principles and thinking tools, and the result is shown in Table 1. In the table thinking tools are labeled with a (T) and principles are labeled with (P) followed by the number and name of the principle or thinking tool as defined in [1]. Principles and tools to the right depend on principles and tools to the left. As part of the analysis it was also observed that the principles and tools addressed different perspectives. Some focused on the engineering practices, another part on the management perspective, and the rest focused on people. These categories are shown as rows below. Finally, the column headers are inspired from Womack and Jones book on "Lean Thinking" where Lean is defined in 5 steps: Value, Value Stream, Flow, Pull and Perfection, see [5].

	Value	**Flow**	**Pull**	**Perfection**
Engineering	*P6 Integrity* T19 Refactor T20 Test	*P2 Amplify Learning* T5 Synchronization T4 Iterations	*P2 Amplify Learning* T3 Feedback T6 Setbased develop.	*P6 Integrity* T18 Conceptual T17 Perceived
Management	*P1 Create Value* T1 Find Waste T2 Value Stream	*P4 Deliver Fast* T11 Queue Theory T12 Cost of delay	*P7 See the Whole* T22 Contracts T21 Measures T10 Pull	*P3 Defer Commitment* T7 Options thinking T8 Defer commitment T9 Decision making
People	*P5 Empower team* T16 Expertise	*P5 Empower team* T14 Motivation	*P5 Empower team* T15 Leadership	*P5 Empower team* T13 Self determination

Table 1 Analysis of causal dependencies in Lean Software Development

The analysis shows that in the engineering dimension, the most dependent parts are the thinking tools "T17 Perceived Integrity" and "T18 Conceptual Integrity". "T18 Conceptual Integrity" is concerned with what is usually understood as good technical quality, whereas "T17 Perceived Integrity" is concerned that the customer receives what he *perceives* to have ordered as opposed to what was *actually* ordered initially.

This thinking tool addresses that the final product fulfills the *actual* needs of the customer and not only the *original* requirement specification. In most cases, the complexity and changes of organizations and technologies introduces the need for an adaptive approach where regular feedback (T3) is needed to identify and absorb the changes necessary to achieve perceived integrity. To allow the customer to provide valuable feedback, the customer needs to see parts of the system as it evolves. In other words, it is necessary that the project is delivered in short iterations (T4) delivering parts of the project, in a way where the customer can relate and comment to it. To deliver high quality code in short iterations, the discipline of test (T20) must be well established, otherwise speed will soon drop as defects are identified and quality is built in afterwards.

The analysis shows similar dependencies for the thinking tools and principles related to Management and People. The model is not used to "implement" Lean, but as a mean to understand the primary tool for future improvements: Lean Software Development.

Using the analysis of Lean Software Development from above and analysis of historical data from all projects in the organization, it was decided to pilot a changed software development process with focus on early test and short iterations.

The improvement resulted in the introduction of a new development method in Systematic named Story Based Development, where work is ultimately decomposed into units called a story. The completion of a story is handled with a checklist, that by design ensures that test of a

story is considered before the story is coded. Another key-aspect is that the work on a story is *inspected* a number of times to ensure that activities are carried out in the right order and are properly completed. The improvement also led to the introduction of Scrum, and it was realized that Scrum addresses many of the thinking tools and principles from Lean Software Development.

The results from adoption of Scrum and Story Based Development were reported at Agile 2007, SEPG2007 and HICCS2008 [2] and how Scrum fit together with other CMMI driven processes were reported at Agile 2008 [3]. The core of these experiences stem from synergies between Lean, CMMI and Scrum.

CMMI provides insight into what processes are needed to maintain a **disciplined** mature organization capable of predicting and improving performance of the organization and projects. Scrum provides guidance for efficient management of projects in a way that allows for high **flexibility** and **adaptability**. When CMMI is interpreted from the basis of Lean values and principles, a large part of project management activities can be implemented using Scrum. When mixing CMMI and Scrum, a magic potion emerges, where the mindset from Scrum ensures that processes are implemented efficiently while embracing change, and CMMI ensures that all relevant processes are considered with proper discipline.

Individually CMMI and Scrum has proven benefits, but also pitfalls. A company can comply with CMMI, but fail to reach optimal performance due to inadequate implementation of processes. Scrum and other agile methods can guide such companies towards more efficient implementation of CMMI.

An agile company may implement Scrum correctly, but fail to obtain the full benefits due to lack of consistent and sufficient execution of engineering or management processes within and across projects in the organization. CMMI can help agile companies to institutionalize agile methods more consistently and understand what processes and disciplines to address.

Systematic has gained valuable experiences in combining Lean, Scrum and CMMI that are relevant both for projects in a CMMI and Scrum context. In particular Systematic can point out where disciplines driven from CMMI, has created a positive synergetic effect to the implementation of Scrum. These activities may not be needed by small or simple projects, but these disciplines are in most cases founded in Lean values, in line with the intentions of Scrum and will be helpful for larger and more complex projects working with Scrum.

3. CMMI, Lean and Scrum combined

Experience. CMMI ensures that agile methods are institutionalized, including consistent implementation throughout the organization, role based training and continuous improvement

We believe the value from Scrum can only be obtained through disciplined use. CMMI has an organizational-level concept of institutionalization that can help establish this needed discipline based on a set of Generic Practices (GP) that are applied to all processes.

Institutionalization is defined in CMMI as “the ingrained way of doing business that an organization follows routinely as part of its corporate culture.” Others have described institutionalization as simply “this is the way we do things around here.” How these GP’s are applied to Scrum can be found in [2]. The GP’s will cover areas like, what is the policy for applying Scrum, how Scrum is performed is documented, relevant responsibility is assigned to Scrum Master and Product Owner, and people fulfilling these roles are trained for the roles.

Institutionalization will ensure that the organization as a whole is committed to Scrum, and that Scrum is performed completely and consistently on different projects, allowing for faster learning and improving of how the organization performs Scrum. Institutionalization will ensure that the Scrum evangelist in each project overall speaks the same gospel.

3.1 Overview

Experience. Scrum also needs some initial planning or setup activities

Experience. Project initiation is fast, and risk are not hidden by speculative planning

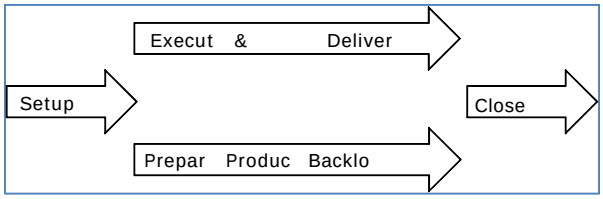


Figure 1 Overall lifecycle model

Using Scrum does not eliminate the need to setup or plan the project initially. Instead initial planning is done with a focus on establishing the project infrastructure, product vision and an initial product backlog based on a mindset to avoid planning based on speculation. This approach reduces the amount of initial planning significantly, but also makes explicit that once the project is setup, it will do two concurrent activities throughout the lifetime of the project.

The part of the Product Backlog that is sufficiently prepared will be the foundation for planning and execution of sprints. The other part of the Product Backlog will iteratively be refined and make more of the product backlog sufficiently prepared for future sprints. Once the whole product backlog is produced, we often find that customers want a final acceptance test of the whole product, even though they have received working code iteratively.

3.2 Setup - Sprint Zero

Experience. Project Planning in CMMI is a disciplined and comprehensive Sprint Zero.

In order to run a good Scrum, it is vital to have good product backlog. CMMI provides very valuable insight into what activities to consider in order to establish the initial Product Backlog during the Project Planning / Sprint Zero. The overall project is planned at a slightly higher level of granularity, and uncertainties are managed with risk management. The primary focus is to establish an initial Product Backlog. The initial Product Backlog is analyzed, and sprints are not initiated until uncertainty and risk are at a level where the projects overall objectives can be met and the draft version of plans and solutions are approved by senior management.

When Systematic adopted Scrum, the project planning process was updated to produce an initial product backlog. Expected CMMI practices include decomposition of work into manageable pieces that are estimated and analyzed for dependencies, planning of stakeholder involvement, and total project risk assessment. In addition to the product backlog a set of overall project plans are established.

Agile teams talk about a sprint zero to establish the foundation for the team to do efficient sprinting. Project Planning in CMMI can be perceived as a sprint zero that also produce a coherent set of plans, that will help improve execution of the product backlog. Such plans cover topics like, stakeholder management, milestone and delivery schedules, cost estimates, and quality.

The initial version of these plans are typically established within few weeks after project initiation and will focus on the most certain elements of the projects plan, leaving more uncertain parts to be elaborated as the project proceeds.

The project planning process, has been changed to primarily deliver a prioritized Product Backlog where the top elements are described in detail, but the lower elements are described in decreasing detail. It is an explicit decision to postpone much planning, and thereby accept that planning and execution will continue concurrently as the project evolves. Many of the CMMI project planning activities are in line with the intentions of Scrum, the main difference is that in CMMI these activities are elaborated and documented.

Experience: Planning and risk management activities reduces risk of product backlog

The first draft product backlog is assessed for risk by the team. Asking the team to estimate the products backlog using 3-point estimates for effort will reveal the most uncertain parts of the work in the Product Backlog. Conducting Risk Identification meetings will identify other important risks, and allow proactive mitigation to be initiated.

Experience. Two levels of planning and tracking: Project as a whole and each sprint.

The project is planned and tracked with plans for the overall project and active sprint(s).

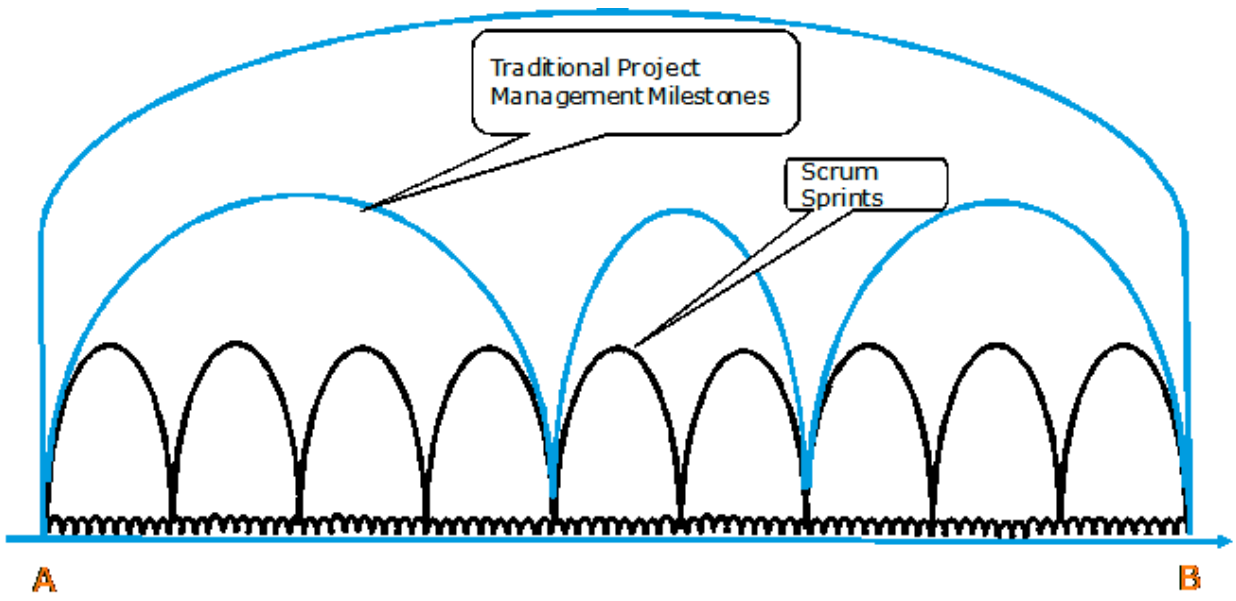


Figure 2 Two levels of planning and tracking

The plans for the overall project established with planning activities during setup provides a better context for defining sprint visions and goals. They allow the team to focus on the sprint, because they can rely on the project manager or Product Owner to track the progress the overall progress.

In general CMMI is used for planning and tracking of the overall project, and Scrum is used for planning and tracking at the sprint level. The CMMI and Scrum activities integrate primarily through the Product Backlog. Sprint Review becomes a heartbeat in tracking of the project and provides high quality feedback into tracking of the overall project plan.

3.3 Execute and Deliver Sprints

Experience: Risk management can proactively prevent impediments

Scrum has a strong focus on removing impediments as soon as they are identified, however CMMI risk management activities focus on proactively identify some of these impediments as risks, and through mitigation eliminate them before they occur as an impediment in the future. The distinction between risk and impediment is that risk describe a problem that may occur, whereas an impediment is problem that has occurred and is impacting planned progress.

Risk management activities are easily integrated with Scrum activities. During project planning the project plans and solution architecture are inputs for initial identification of risks. During project execution, bi-weekly meetings of 10-15 minutes are arranged, where the status of known risks is reported and new risks are identified. It is our experience that these risk management meetings should be kept outside the daily scrum meeting. New risks may be reported on the daily scrum, in which case the risk manager will just take a note.

Experience: Use checklist to ensure quality of stories

One of the important aspects of Systematic Story Based Development method was to ensure focus on early test. The quality of stories are generally ensured by focus on early specification of test and by getting somebody else to look at the work done.

Developing a story includes many different activities, that need to be structured to some degree. The Story Completion Checklist accomplishes these goals, by structuring activities and defining when work must be inspected by an inspector.

Experience: Automated test is a must in order to do one month sprints

When the sprint duration is one month, all tests must be automated to the extent possible. It is an integrated part of developing a story, to also implement the automated test verifying the story. Automated test are used on the teams shared repository and run every time a developer commits code to the shared build server.

Experience: Continuous integration must be supported with discipline for check in

In order to avoid chaos when developers continuously integrate with each other, we have defined the following criteria for check in of code to the integration repository: The test must run smoothly in the developers sandbox and the code must comply with the code standard checked with FxCop (a static code analysis tool).

Experience: Automated test and integration must be supported by a standard production line

Every project needs this infrastructure and therefore we have established standard production line setups, allowing projects to get started faster. Scrum promotes short iterations, e.g. one sprint per month, and in our experience this drives the need for a strong discipline in configuration management, test, integration and release. You can only deliver high quality sprint deliveries every month, if you address defects immediately they are identified as opposed to store and fix them later. In a 1 month sprint you should spent at most 3 calendar days to test, integrate and release. This is only possible if you build quality in, and test is expected to NOT find any defects.

To establish this capability CMMI helps with:

- Establish standards for production line, including standard setups for build- and test servers
- Establish discipline on criteria for integration
- Measures to objectively evaluate performance
- Disciplines to maintain integrity of configuration management system, builds, and releases.

Experience: CMMI facilitates data driven identification and elimination of impediments

From a CMMI perspective Scrum is one process out of a set of processes used to execute a project. In a CMMI context all processes for development are monitored for effectiveness and efficiency. Therefore measures are also established on the Scrum process. The choice of measures were inspired from Lean [3] and from the objective to establish a stable flow of work.

We wanted a measure to help establish focus on a “Stop the line” mindset to defects, to ensure defects are addressed immediately after they are identified. We also wanted insight into the flow of story implementation – that is, how much waiting time is incurred when a story is implemented.

These considerations led to a number of measures where the most important are:

1. Fix time for failed builds – are problems proactively handled?
2. Flow in implementation of story – is a story implemented without breaks in calendar time and context shift to implementation of other stories?

The main reason to measure how long it takes from a build fails on the shared build server until the next succeeding build has to do with speed and quality. If a defect or a problem is not addressed immediately after it is identified, rework will accumulate and it will be difficult to deliver a sprint with high quality and maintain a high velocity. The two measures can be considered a sub-process for fixing failed builds and coding a story respectively. The measures are regularly analyzed using statistical control charts, and data-points outside the control-limits are evaluated. Analysis of such data often identifies project impediments.

Experience: Focus on "fix-time after failed build" drives good discipline in project

Using standard infrastructure setups allows for efficient data collection and analysis. In particular Systematic has been inspired from Lean thoughts on flow and jidoka (stopping the production line when an error is detected).

We want our projects to be able to deliver on a daily basis, and hence that unresolved failed builds are fixed within a working day. We use CruiseControl (a build management tool) to signal all developers when a build fails. We also monitor the objective by analyzing data from the build servers using control charts like the one shown below.

Many projects have achieved this one work day objective, merely by the focus on the measure. The objective is easy to understand, and presenting the information in CruiseControl and control charts has established a good habit of fixing broken builds immediately when they fail.

3.4 Prepare Product Backlog

Experience: Explicit quality plans helps the team to build the right quality in

One of the results of planning is a quality assurance schedule (QAS), where it is outlined what quality activities will be used to ensure the quality objectives are achieved and when. The QAS may specify

- What stories are subject to inspection
- What code is subject to review
- What documents are subject to what types of review
- What unit test and automatic test is produced
- What is included in the acceptance test

A typical QAS document is only a few pages long, but the above descriptions help a scrum team to elaborate and understand the definition of done.

Experience The Product Backlog must be groomed with at least the same speed as sprints are delivered

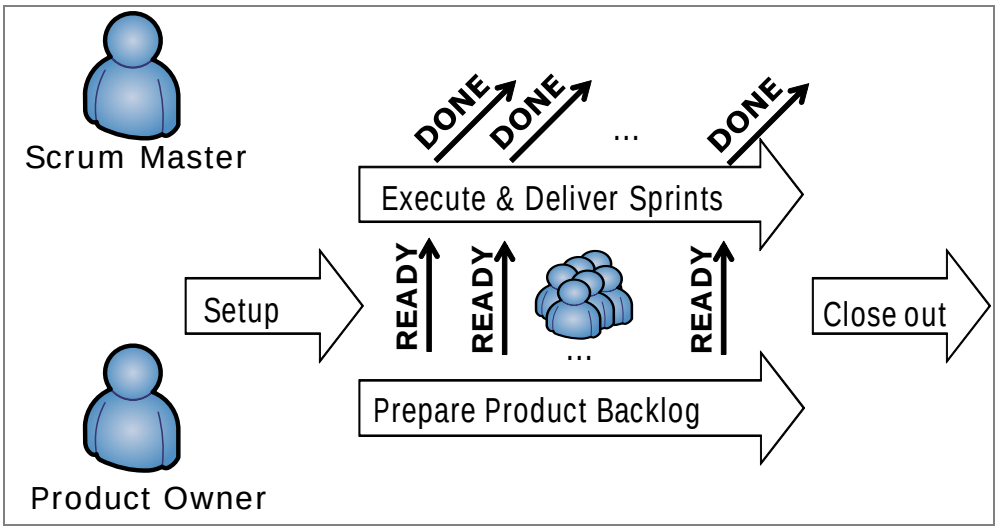


Figure 3 Balancing preparing and delivering activities

The general experiences from working with CMMI and Scrum is that it requires strong discipline to deliver high quality code after each sprint, while at the same time ensure sufficient planning activities to groom the Product Backlog.

The initial plan ensures a sufficient confidence for the project as a whole, and postpones detailed planning to be executed as the project executes. The activities postponed often include clarification or agreement to interpretation of requirements. The challenge is that these activities often requires participation from members of the sprint team, who are committed to deliver a sprint delivery.

The amount of time spent preparing features on the product backlog will vary over time, and it is difficult to make a fixed plan for how much time to spend. Instead the project track how much work is ready for future sprints concurrent to tracking the burn down chart for current sprint. This provides a visual feedback showing if planning activities are starved.

The reason we talk about whether a feature or story is ready for implementation, is that we inspired from Lean want to have a smooth flow of implementing stories or features from the backlog. We found that one of the main impediments for a smooth flow when executing sprints, was that the work allocated to the sprint was not sufficiently prepared or clarified. Therefore we will only allocate features or stories to a sprint when they are sufficiently clarified. Based on our CMMI implementation it is well defined when a feature is READY, and the work to bring from an initial state to the state READY is supported by a “Feature Ready for Implementation Checklist”.

The Product Owner is responsible for ensuring that the Product Backlog is constantly groomed to keep at least an amount of 110% sprint velocity READY on the top of the Product Backlog. This assures that new sprints can be planned, and only will allocate work that is READY. By doing so we achieved significant better flow of story implementation in our projects.

Assume a story is estimated to be 3 workdays of effort. However for various reasons it takes 9 workdays to implement the story. The flow of this story implementation is defined as 3 days calendar time of work implemented over 9 calendar days, a flow of 3/9 or 33%.

When we started measuring flow of story implementation it was around 30%, from 2007 to 2008 it increased to approximately 60% for Q4 2008. Efficient flow eliminates the waste

associated with context shifts and handovers. In addition the team members find it more satisfying that work initiated in a sprint is sufficiently clarified to allow for a smooth implementation during the sprint.

4. Recommendations

Systematic has experienced synergies between Lean, CMMI and Scrum that scales Scrum to be successful in larger or distributed projects and organizations. The experiences from Systematic can be turned into the a number of recommendations that may help you to a great Scrum. The recommendations are presented in the context of a stepwise adoption model for Scrum, that we believe many projects go through. Projects inexperienced with Scrum are often seen to only adopt parts of Scrum. This may have little or even negative effect. As Scrum is fully implemented and the projects build experience, they learn how to bring the full benefits out of Scrum, and will typically improve significantly. Instead of discussing the quantity of these gains, we will focus on what activities or disciplines we typically see a project master, as it brings the full benefit out of Scrum:

Scrum Type	Driven by
ScrumButt	Lack of discipline to implement all of Scrum
Pretty Good Scrum	Basic Scrum roles and meetings established, teams co-located and values mutual respect and shared commitment to sprint goal.
Good Scrum	Discipline in execution of Sprint and using DONE concept
Great Scrum	Discipline in grooming the product backlog using READY concept

4.1 ScrumButt

In order to avoid a partial Scrum implementation, Systematic have one interpretation of how to execute Scrum shared by all projects in Systematic and has gradually standardized on concepts. For example all projects are recommended to use a Physical Scrum Board for the sprint backlog and not a particular tool.

1. Use CMMI Generic Practices to ensure a full and consistent implementation of Scrum in all projects in your organization including training of Scrum Master and Product Owner to their roles. How this is done is elaborated in [2] and [3].
2. Use the Nokia test to identify gabs in your Scrum implementation, for specific descriptions and guidelines please visit http://jeffsutherland.com/scrum/nokiatest.pdf

4.2 Pretty Good Scrum

3. Co-locate project and teams to the extent possible
4. Establish your own sprint zero, and include activities in the items below
5. Use Risk Management to proactively address risks before they are identified as impediments
6. Transform requirements into a solution description described in a set of high level features on the product backlog. Prepare the product backlog by decomposing the highest prioritized features to stories allowing for efficient sprint planning.
7. Use 3-point effort estimates on elements of the product backlog during initial planning.
8. Analyze dependencies, stakeholders, risk on elements of product backlog.
9. Establish milestone and delivery plan (roadmap) and their initial relationship to product backlog.

4.3 Good Scrum

10. Decide and communicate on quality objectives including, what code and documentation to formally review to elaborate your projects definition of DONE.
11. Use Story Completion Checklist and inspection to maintain high quality of stories produced and ensure that DONE criteria is met
12. Establish standards for project "production line" including development, build servers, and test servers.
13. Automate test and nightly build, and measure performance.
14. Establish criteria for committing of code to integration.
15. Maintain integrity of configuration management
16. Use a checklist for Work Product Evaluation to ensure and validate that sprint deliveries are consistent.

4.4 Great Scrum

17. Define READY criteria for work on the product backlog.
18. Ensure visibility of the amount of work on the product backlog that is ready for Sprint Planning. The amount of READY work should at all times be kept to at least Sprint velocity of all teams
19. Product Owners work is not included in sprint plans but sprint is planned to support that team members can provide sufficient support to Product Owner activities.

5. Conclusion

Systematic has worked with Lean, CMMI and Scrum for several years, and found that they can be mixed in a way where strong synergies are achieved. The focus of this article has been to share how Lean and CMMI can bring synergies and support your adoption of Scrum and agile. The experiences were presented as recommendations for agile projects on activities or disciplines to adopt particularly in larger or distributed projects. These recommendations are based on experiences gained from projects at Systematic running Scrum in the context of Lean and CMMI. We believe that these experiences can be adopted by many agile projects, and are independent to whether your project or organization is working according to CMMI or Lean.

The experience from projects at Systematic is that to achieve the full benefit of Scrum you should use the concept of READY and DONE. Systematic identified the need for and value of the READY criteria, in our search for stable flow in Sprints. Adopting the READY concept makes it clear that the Product Backlog is divided into the following items:

1. In Sprint (work in progress – facilitated by Scrum Master controlled with DONE criteria)
2. Ready (Prioritized list of work ready for future Sprint Planning Meetings)
3. Preparing (work in progress – facilitated by Product Owner controlled by READY criteria)
4. New (Prioritized list of work waiting to be made ready)

It is important to understand that READY is the least amount of activity bringing an unspecified high granularity item on the backlog, into a state where it is <u>sufficiently</u> described to allow the team to implement the item without disruption.

Finally we advice other agile projects to map themselves against the Scrum adoption model presented above, and to apply the above recommendations stepwise from ScrumButt to Great Scrum.

References

[1] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Implementation Guide*: Addison-Wesley, 2006.

[2] J. Sutherland, C.R. Jakobsen and K.A. Johnson, "CMMI and Scrum - a magic potion for code warriors" in proceedings for Agile 2007

[3] C.R. Jakobsen and K. A. Johnson, “Agile with a twist of CMMI” in proceedings for Agile 2008

[4] M. K. Kulpa and K. A. Johnson, *Interpreting the CMMI: A Process Improvement Approach, Second Edition*. Boca Raton: Auerbach Publications, 2008

[5] Womack and Jones, *Lean Thinking: Banish Waste and Create Wealth in your Cooperation*, Free Press, 2003.

Implementing Automated Software Testing
Continuously Track Progress - and Adjust Accordingly

Thom Garrett, tgarrett@idtus.com
Innovative Defense Technologies, www.IDTus.com

"When you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind." - Lord Kelvin

Most of us have worked on at least one project where the best-laid plans went awry and at times there wasn't just one reason we could point to that caused the failure. People, schedules, processes, and budgets can all contribute. [1] Based on such experiences from the past, we have learned that as part of a successful automated testing (AST) program it is important that the right people with the applicable skills are hired, goals and strategies be defined and then implemented, and that steps be put in place to continuously track, measure, and adjust, as needed, against these goals and strategies. Here we'll discuss the importance of tracking the AST program, to include various defect prevention techniques, such as peer reviews and other interchanges. We'll then focus on the types of AST metrics to gather so that we can measure progress, gauge the effectiveness of our AST efforts, and help keep them keep on track and/or make adjustments, if necessary. Finally, we will discuss the importance of a root/cause analysis if a defect or issue is encountered.

Based on the outcome of these various efforts, adjustments can be made where necessary; e.g. the defects remaining to be fixed in a testing cycle can be assessed, schedules can be adjusted, and/or goals can be reduced. For example, if a feature is left with too many high-priority defects, a decision can be made to move the ship date, to ship the system as is (which generally isn't wise, unless a quick patch process is in place), or to go live without that specific feature, if that is feasible.

Success is measured based on achieving the goals we set out to accomplish relative to the expectations of our stakeholders and customers.

AST Program Tracking and Defect Prevention

In our book "Implementing Automated Software Testing (IAST)", we cover the importance of valid requirements and their assessment; in another we discuss the precautions to take when deciding what to automate; and yet another section we discussed in detail the importance of peer reviews. Here we'll provide additional ideas that aid in defect prevention efforts, including technical interchanges and walk-throughs; internal inspections; examination of constraints and associated risks; risk mitigation strategies; safeguarding AST processes and environments via configuration management; and defining and tracking schedules, costs, action items, and issues/defects.

Conduct Technical Interchanges and Walk-throughs

Peer reviews, technical interchanges, and walk-throughs with the customer and the internal AST team represent evaluation techniques that should take place throughout the AST effort. These techniques can be applied to all AST deliverables - test requirements, test cases, AST design and

code, and other software work products, such as test procedures and automated test scripts. They consist of a detailed artifact examination by a person or a group other than the author. These interchanges and walk-throughs are intended to detect defects, non-adherence to AST guidelines, test procedure issues, and other problems.

An example of a technical interchange meeting is an overview of test requirement documentation. When AST test requirements are defined in terms that are testable and correct, errors are prevented from entering the AST development pipeline that could eventually be reflected as defects in the deliverable. AST design component walk-throughs can be performed to ensure that the design is consistent with defined requirements - e.g. that it conforms to OA standards and applicable design methodology - and that errors are minimized.

Technical reviews and inspections have proven to be the most effective forms of preventing miscommunication, allowing for defect detection and removal.

Conduct Internal Inspections

In addition to customer technical interchanges and walk-throughs, internal inspections of deliverable work products should take place, before anything is even presented to the customer, to support the detection and removal of defects and process/practice omissions or deficiencies early in the AST development and test cycle; prevent the migration of defects to later phases; improve quality and productivity; and reduce cost, cycle time, and maintenance efforts.

Examine Constraints and Associated Risks

A careful examination of goals and constraints and associated risks should take place, leading to a systematic AST strategy and producing a predictable, higher-quality outcome and a high degree of success. Combining a careful examination of constraints together with defect detection technologies will yield the best results.

Any constraint and associated risk should be communicated to the customer and risk mitigation strategies developed as necessary.

Implement Risk Mitigation Strategies

Defined "light weight" processes allow for iterative, constant risk assessment and review without the dreaded overhead. If a risk is identified, appropriate mitigation strategies can be deployed. Require ongoing review of cost, schedules, processes, and implementation to ensure that potential problems do not go unnoticed until too late. Instead, processes need to ensure that problems are addressed and corrected immediately. For example, how will you mitigate the risk if your "star" developer quits? There are numerous possible answers: Software development is a team effort and it is never a good practice to rely on one "star" developer. Hire qualified developers, so they can integrate as a team and each can be relied on in various ways based on their respective qualifications. One team member might have more experience than another, but neither should be irreplaceable, and the departure of one of them should not be detrimental to the project. Follow good hiring and software development practices (such as documenting and maintaining all AST-related artifacts) and put the right people on the project; we discuss the "how to" in our book "IAST." Additional risks could be missed deadlines or being over budget. Evaluate and determine risk mitigation techniques in case an identified risk comes to fruition.

Safeguard the Integrity of the AST Process and Environments

Experience shows that it is important to safeguard the integrity of the AST processes and environment. In IAST we discuss the importance of an isolated test environment and having it under configuration management. For example, you might want to test any new technology to be used as part of the AST effort in an isolated environment and validate that a tool, for example, performs to product specifications and marketing claims before it is used on any AUT or customer test environment. At one point we installed a tool on our Micron PC used for daily activities, only to have it blue-screen. It turned out that the tool we wanted to test wasn't compatible with the Micron PC. To solve the problem, we actually had to upgrade the PC's BIOS. An isolated test environment for these types of evaluation activities is vital.

The automator should also verify that any upgrades to a technology still run in the current environment. The previous version of the tool may have performed correctly and a new upgrade may perform fine in other environments, but the upgrade may adversely affect the team's particular environment. We had an experience when a new tool upgrade wasn't compatible with our e-mail software package any longer. It was a good thing we caught this issue, because otherwise an upgrade install would have rendered the tool useless, as we heavily relied on e-mail notification, for example, if a defect was generated.

Additionally, using a configuration management tool to baseline the test repository will help safeguard the integrity of the automated testing process. For example, all AST automation framework components, script files, test case and test procedure documentation, schedules, cost tracking, and other related AST artifacts need to be under configuration management. Using a configuration management tool ensures that the latest and most accurate version control and

records of AST artifacts and products are maintained. For example, we are using the open-source tool Subversion in order to maintain AST product integrity. We evaluate the best products available to allow for the most efficient controls on an ongoing basis.

Define, Communicate, and Track Schedules and Costs

It is not good enough to base a schedule on a marketing-department-defined deadline. Instead, schedule and task durations need to be determined based on past historical performance and associated best estimates gathered from all stakeholders. Additionally, any schedule dependencies and critical path elements need to be considered up front and incorporated into the schedule. Project schedules need to be defined, continuously tracked, and communicated.

In order to meet any schedule—for example, if the program is under a tight deadline - only the AST tasks that can be successfully delivered in time are included in the schedule iteration. As described in IAST, during the AST Phase 1, test requirements are prioritized, which allows for prioritizing the most critical AST tasks to be completed as opposed to the less critical and lower-priority tasks, which can then be moved to later in the schedule, accordingly. Once the requirements are prioritized, an initial schedule is presented to the customer for approval and not before the System Under Test (SUT), AST requirements and associated level of effort are understood.

During the technical interchanges and walk-throughs, schedules are evaluated and presented on an ongoing basis to allow for continuous communication and monitoring. Potential schedule risks should be communicated well in advance and risk mitigation strategies explored and implemented, as needed. Any schedule slips should be communicated to the customer immediately and adjustments made accordingly.

By closely tracking schedules and other required AST resources, we can also ensure that a cost tracking and control process is followed. Inspections, walk-throughs, and other status reporting allow for closely monitored cost control and tracking. Tracking cost and schedules and so forth allows for tracking of the project's performance.

Track Actions, Issues, and Defects

A detailed procedure needs to be defined for tracking action items to completion. Templates should be used that describe all elements to be filled out for action item reports.

Additionally, a procedure needs to be in place that allows for tracking issues/defects to closure, known as a defect tracking lifecycle. See IAST for a sample defect tracking lifecycle used in the open-source defect tracking tool Bugzilla. Various defect tracking lifecycles exist; adapt one to your environment, tool, and project needs. Once defined, put measures in place to verify that the defect or action item lifecycle is adhered to.

If an issue or defect is uncovered, a root cause analysis should be conducted. See that section later on for more on root cause analysis.

AST Metrics

Metrics can aid in improving your organization's automated testing process and tracking its status. Much has been said and written about the need for using metrics carefully and to not let metrics drive an effort, i.e. don't measure for the sake of measuring. As with our recommended lightweight and adjustable process described in IAST, we recommend to use these metrics as an enhancement to the AST effort not to drive the AST effort. Our software test teams have successfully used the metrics and techniques discussed here. As the beginning quote implies, if you can measure something, then you have something you can quantify.

As time proceeds, software projects become more complex because of increased lines of code as a result of added features, bug fixes, etc. Also, tasks must be done in less time and with fewer people. Complexity over time has a tendency to decrease the test coverage and ultimately affect the quality of the product. Other factors involved over time are the overall cost of the product and the time in which to deliver the software. Carefully defined metrics can provide insight into the status of automated testing efforts.

When implemented properly, AST can help reverse the negative trend. As represented in Figure 1.1, automation efforts can provide a larger test coverage area and increase the overall quality of a product. The figure illustrates that the goal of automation is ultimately to reduce the time of testing and the cost of delivery, while increasing test coverage and quality. These benefits are typically realized over multiple test and project release cycles.

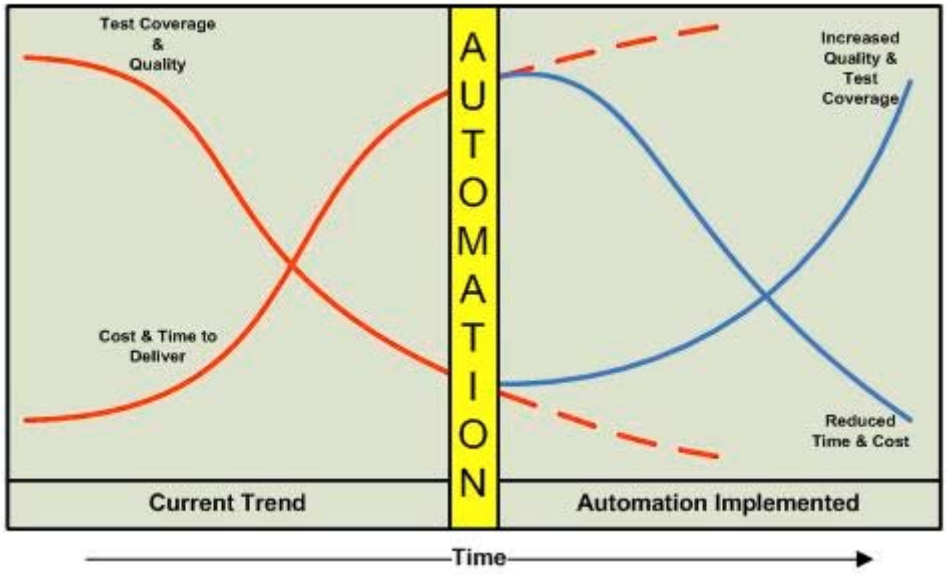


Figure 1.1 AST goal examples comparing current trend with automation implementation

Automated testing metrics can aid in making assessments as to whether coverage, progress and quality goals are being met. Before we discuss how these goals can be accomplished, we want to define metrics, automated testing metrics, and what makes a good automated test metric.

What is a metric? The basic definition of a *metric* is a standard of measurement. It can also be described as a system of related measures that facilitate the quantification of some particular characteristic. [2] For our purposes, a metric can be seen as a measure that can be used to display past and present performance and/or predict future performance.

Metrics categories: Most software testing metrics (including the ones presented here) fall into one of three categories:

- **Coverage:** meaningful parameters for measuring test scope and success.
- **Progress:** parameters that help identify test progress to be matched against success criteria. Progress metrics are collected iteratively over time. They can be used to graph the process itself (e.g., time to fix defects, time to test, etc.).
- **Quality:** meaningful measures of testing product quality. Usability, performance, scalability, overall customer satisfaction, and defects reported are a few examples.

What are automated testing metrics? Automated testing metrics are metrics used to measure the performance (past, present, and future) of the implemented automated testing process and related efforts and artifacts. Here we can also differentiate metrics related to unit test automation versus integration or system test automation. Automated testing metrics serve to enhance and complement general testing metrics, providing a measure of the AST coverage, progress, and quality, not replace them.

What makes a good automated testing metric? As with any metrics, automated testing metrics should have clearly defined goals for the automation effort. It serves no purpose to measure something for the sake of measuring. To be meaningful, a metric should relate to the performance of the effort.

Prior to defining the automated testing metrics, there are metrics-setting fundamentals you may want to review. Before measuring anything, set goals. What is it you are trying to accomplish? Goals are important; if you do not have them, what is it that you are measuring? It is also important to track and measure on an ongoing basis. Based on the metrics outcome, you can decide whether changes to deadlines, feature lists, process strategies, etc., need to be adjusted accordingly. As a step toward goal setting, questions may need to be asked about the current state of affairs. Decide what questions to ask to determine whether or not you are tracking toward the defined goals. For example:

- How many permutations of the test(s) selected do we run?
- How much time does it take to run all the tests?
- How is test coverage defined? Are we measuring test cases against requirements (generally during system testing), or are we measuring test cases against all possible paths taken through the units and components (generally used for unit testing)? In other words, are we looking at unit testing coverage, code coverage, or requirements coverage?
- How much time does it take to do data analysis? Are we better off automating that analysis? What would be involved in generating the automated analysis?
- How long does it take to build a scenario and required driver?
- How often do we run the test(s) selected?
- How many people do we require to run the test(s) selected?
- How much system and lab time is required to run the test(s) selected?

In essence, a good automated testing metric has the following characteristics:

- It is objective.
- It is measurable.
- It is meaningful.
- Data for it is easily gathered.
- It can help identify areas of test automation improvement.
- It is simple.

A few more words about metrics being simple: Albert Einstein once said, “Make everything as simple as possible, but not simpler.” When applying this wisdom to AST and related metrics collection, you will see that

- Simplicity reduces errors.
- Simplicity is more effective.
- Simplicity is elegant.
- Simplicity brings focus.

It is important to generate a metric that calculates the value of automation, especially if this is the first time an automated testing approach has been used for a project. IAST discusses ROI measurement in detail and provides various worksheets that can serve as a baseline for calculating AST ROI. For example, there we mention that the test team will need to measure the time spent on developing and executing test scripts against the results that the scripts produce. If needed, the test team could justify the number of hours required to develop and execute AST by

providing the number of defects found using this automation that would likely not have been revealed during a manual test effort. Specific details as to why the manual effort would not have found the defect can be provided; some possible reasons are that the automated test used additional test data not previously included in the manual effort, or the automated test used additional scenarios and path coverage previously not touched manually. Another way of putting this is that, for example, with manual testing you might have been able to test x number of test data combinations; with automated testing you are now able to test $x + y$ test data combinations. Defects that were uncovered in the set of y combinations are the defects that manual testing may have never uncovered. Here you can also show the increase in testing coverage for future software releases.

Another way to quantify or measure automation benefits is to show that a specific automated test could hardly have been accomplished in a manual fashion. For example, say that during stress testing 1,000 virtual users execute a specific functionality and the system crashes. It would be very difficult to discover this problem manually, using 1,000 test engineers or possibly even extrapolation as it is still very commonly used today.

AST can also minimize the test effort, for example, by the use of an automated test tool for data entry or record setup. Consider the test effort associated with the system requirement that reads, "The system shall allow the addition of 10,000 new accounts." Imagine having to manually enter 10,000 accounts into a system in order to test this requirement! An automated test script can easily support this requirement by reading account information from a file through the use of a looping construct. The data file can easily be generated using a data generator. The effort to verify this system requirement using test automation requires far fewer man-hours than performing such a test using manual test methods. [3] The ROI metric that applies in this case measures the time required to manually set up the needed records versus the time required to set up the records using an automated tool.

What follows are additional metrics that can be used to help track progress of the AST program. Here we can differentiate between test case and progress metrics and defect and defect removal metrics.

Percent Automatable or Automation Index

As part of an AST effort, the project is either basing its automation on existing manual test procedures, or starting a new automation effort from scratch, some combination, or even just maintaining an AST effort. Whatever the case, a percent automatable metric or the automation index can be determined.

Percent automatable can be defined as the percentage of a set of given test cases that is automatable. This could be represented by the following equation:

$$PA\ (\%) = \frac{ATC}{TC} = \left(\frac{\textit{No. of test cases automatable}}{\textit{Total no. of test cases}} \right)$$

PA = Percent automatable
ATC = Number of test cases automatable
TC = Total number of test cases

When evaluating test cases to be developed, what is to be considered automatable and what is to be considered not automatable? Given enough ingenuity and resources, one can argue that

almost anything can be automated. So where do you draw the line? Something that can be considered "not automatable," for example, could be an application area that is still under design, not very stable, and mostly in flux. In cases such as this, you should evaluate whether it makes sense to automate. See IAST for a detailed discussion of how to determine what to automate. There we discussed that we would evaluate, given the set of test cases, which ones would provide the biggest return on investment if automated. **Just because a test is automatable doesn't necessary mean it should be automated.**

Prioritize your automation effort based on the outcome of this "what to automate" evaluation. Figure 1.2 shows how this metric can be used to summarize, for example, the percent automatable of various projects or components within a project and to set the automation goal. Once we know the percent automatable, we can use it as a baseline for measuring AST implementation progress.

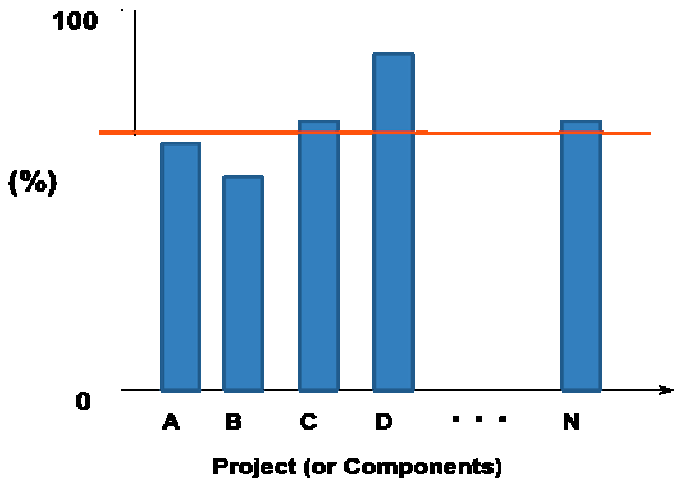


Figure 1.2 Example of percent automatable (automation index) per project (or component)

Automation Progress

Automation progress refers to the number of tests that have been automated as a percentage of all automatable test cases. Basically, how well are you doing against the goal of automated testing? The ultimate goal is to automate 100% of the "automatable" test cases. This can be accomplished in phases, so it is important to set a goal that states the deadlines for when a specific percentage of the ASTs should be automated. It is useful to track this metric during the various stages of automated testing development.

$$AP\ (\%) = \frac{AA}{ATC} = \left(\frac{\textit{No. of test cases automated}}{\textit{No. of test cases automatable}} \right)$$

AP = Automation progress
AA = Number of test cases automated
ATC = Number of test cases automatable

The automation progress metric is a metric typically tracked over time. In the case of Figure 1.3, the time is weeks.

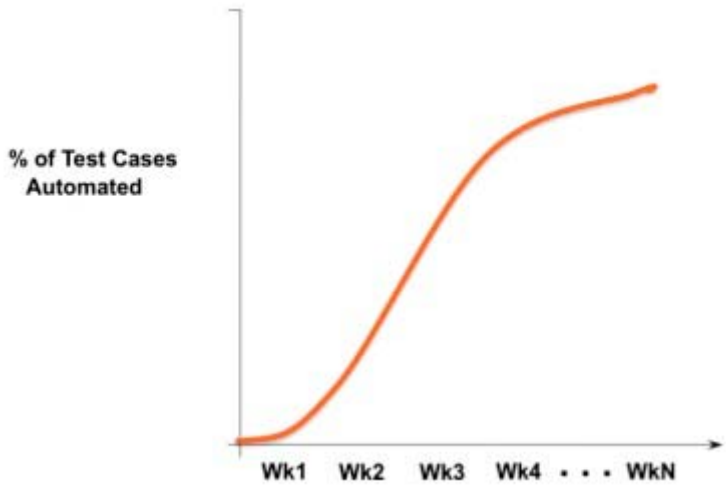


Figure 1.3 Test cases automated over time (weeks)

Test Progress

A common metric closely associated with the progress of automation, yet not exclusive to automation, is *test progress*. Test progress can simply be defined as the number of test cases (manual and automated) executed over time.

$$TP = \frac{TC}{T} = \left(\frac{\textit{No. of test cases executed}}{\textit{Total number of test cases}} \right)$$

TP = Test progress
TC = Number of test cases executed
T = Total number of test cases

The purpose of this metric is to track test progress and can be used to show how testing is tracking against the overall project plan.

More detailed analysis is needed to determine test pass/fail, which can be captured in a more refined metric; i.e., we need to determine not only how many tests have been run over time and how many more there are to be run, but also how many of those test executions actually pass consistently without failure so that the test can actually be considered complete. In the test progress metric we can replace *No. of test cases executed* with *No. of test cases completed*, counting only those test cases that actually consistently pass.

Percent of Automated Test Coverage

Another AST metric we want to consider is *percent of automated test coverage*. This metric determines what percentage of test coverage the automated testing is actually achieving. Various degrees of test coverage can be achieved, depending on the project and defined goals. Also depending on the types of testing performed, unit test automation coverage could be measured against all identified units, or functional system test coverage can be measured against all requirements, and so forth. Together with manual test coverage, this metric measures the completeness of the test coverage and can measure how much automation is being executed relative to the total number of tests. However, it does not say anything about the quality of the automation. For example, 2,000 test cases executing the same or similar data paths may take a

lot of time and effort to execute, but they do not equate to a larger percentage of test coverage. Test data techniques discussed in IAST need to be used to effectively derive the number of test data elements required to test the same or similar data path. Percent of automated test coverage does not indicate anything about the effectiveness of the testing taking place; it is a metric that measures its dimension.

$$PTC\ (\%) = \frac{AC}{C} = \left(\frac{\textit{Automation coverage}}{\textit{Total coverage}} \right)$$

PTC = Percent of automated test coverage
AC = Automation coverage
C = Total coverage (i.e., requirements, units/components, or code coverage)

There is a wealth of material available regarding the sizing or coverage of systems. A useful resource is Stephen H. Kan's book *Metrics and Models in Software Quality Engineering*. [4]

Figure 1.4 provides an example of test coverage for Project A versus Project B over various iterations. The dip in coverage for Project A might reveal that new functionality was delivered that hadn't yet been tested, so that no coverage was provided for that area.

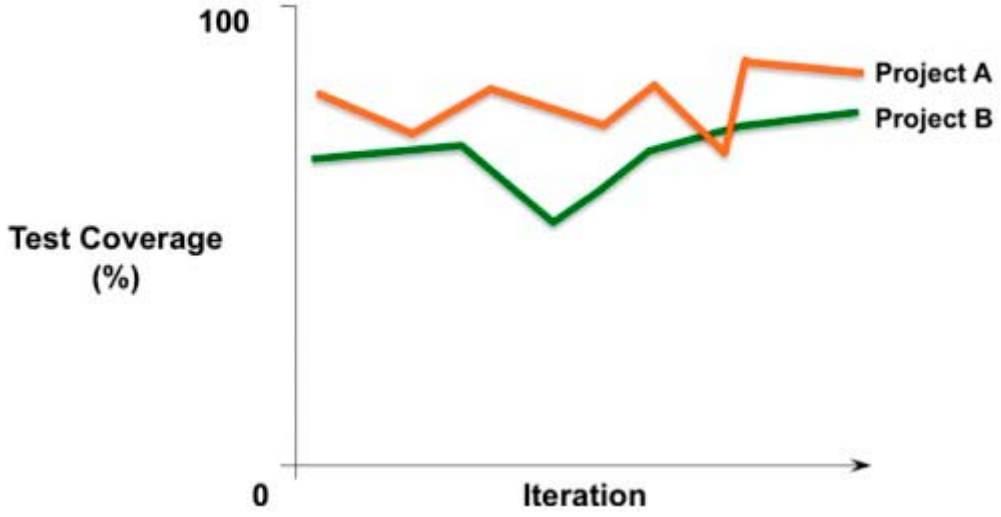


Figure 1.4 Test coverage per project over various iterations

Defect Density

Measuring defects is a discipline to be implemented regardless of whether the testing effort is automated or not. *Defect density* is another well-known metric that can be used for determining an area to automate. If a component requires a lot of retesting because the defect density is very high, it might lend itself perfectly to automated testing. Defect density is a measure of the total known defects divided by the size of the software entity being measured. For example, if there is a high defect density in a specific functionality, it is important to conduct a causal analysis. Is this functionality very complex, and therefore is it to be expected that the defect density would be high? Is there a problem with the design or implementation of the functionality? Were the wrong (or not enough) resources assigned to the functionality, because an inaccurate risk had been assigned to it and the complexity was not understood? It also could be inferred that the developer responsible for this specific functionality needs more training.

$$DD = \frac{D}{SS} = \left(\frac{\text{No. of known defects}}{\text{Size of software entity}} \right)$$

DD = Defect density
D = Number of known defects
SS = Size of software entity

We can't necessary blame a high defect density on large software component size, i.e. while generally the thought is that a high defect density in a large component is more justifiable than in a small component, the small component could be much more complex than the large one. AST complexity is an important consideration when evaluating defect density.Additionally, when evaluating defect density, the priority of the defect should be considered. For example, one application requirement may have as many as 50 low-priority defects and still pass because the acceptance criteria have been satisfied. Still another requirement may have only one open defect, but that defect prevents the acceptance criteria from being satisfied because it is a high priority. Higher-priority defects are generally weighted more heavily as part of this metric.

Defect Trend Analysis

Another useful testing metric in general is *defect trend analysis*. Defect trend analysis is calculated as:

$$DTA = \frac{D}{TPE} = \left(\frac{\text{No. of known defects}}{\text{No. of test procedures executed}} \right)$$

DTA = Defect trend analysis
D = Number of known defects
TPE = Number of test procedures executed over time

Defect trend analysis can help determine the trend of defects found over time. Is the trend improving as the testing phase is winding down, or does the trend remain static or is it worsening? During the AST testing process, we have found defect trend analysis to be one of the more useful metrics to show the health of a project. One approach to showing the trend is to plot the total number of defects over time, as shown in Figure 1.5. [5]

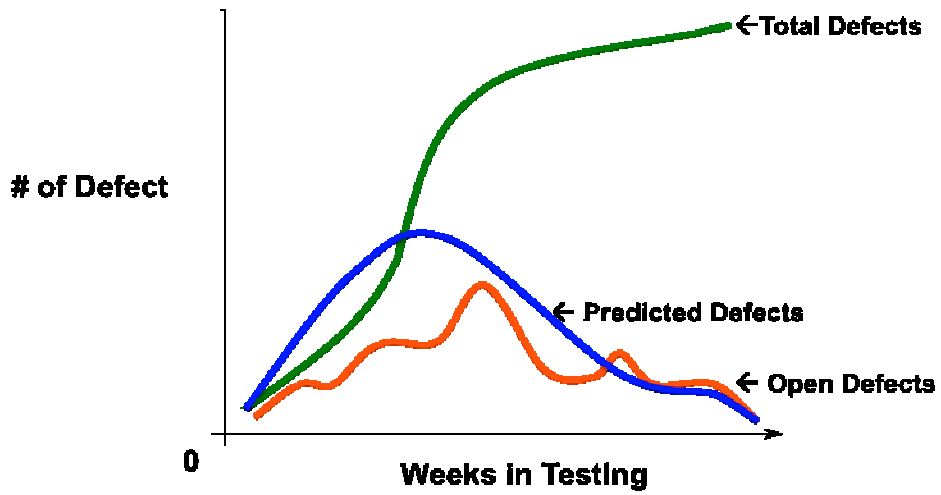


Figure 1.5 Defect trend analysis: total number of defects over time (here after weeks in testing)

Effective defect tracking analysis can present a clear view of the status of testing throughout the project.

Defect Removal Efficiency

One of the more popular metrics is *defect removal efficiency* (DRE); this metric is not specific to automation, but it is very useful when used in conjunction with automation efforts. DRE is used to determine the effectiveness of defect removal efforts. It is also an indirect measurement of product quality. The value of the DRE is calculated as a percentage. The higher the percentage, the greater the potential positive impact on the quality of the product. This is because it represents the timely identification and removal of defects at any particular phase.

$$DRE\ (\%) = \frac{DT}{DT + DA} = \left(\frac{\textit{No. of defects found during testing}}{\textit{No. of defects found during testing} + \textit{No. of defects found after delivery}} \right)$$

DRE = Defect removal efficiency
DT = Number of defects found during testing
DA = Number of defects found after delivery

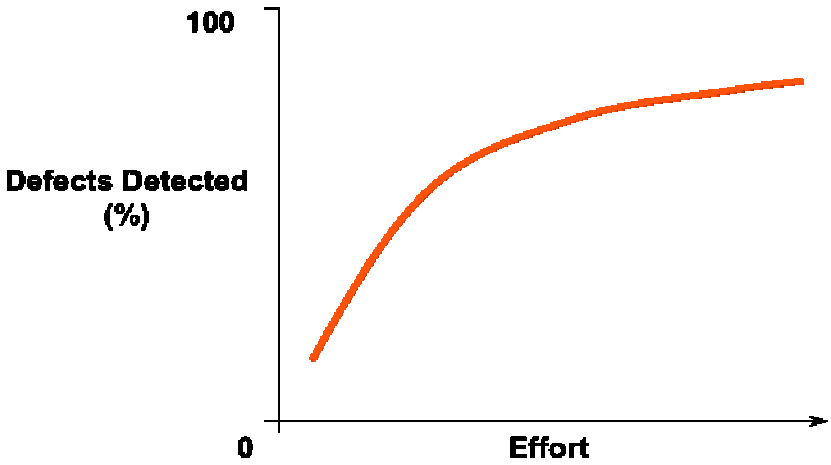


Figure 1.6 Defect removal efficiency over the testing lifecycle effort

The highest attainable value of DRE is 1, which equates to 100%. In practice, we have found that an efficiency rating of 100% is not likely. According to Capers Jones, world-class organizations have a DRE greater than 95%. [6] DRE should be measured during the different development phases. For example, low DRE during analysis and design may indicate that more time should be spent improving the way formal technical reviews are conducted. This calculation can be extended for released products as a measure of the number of defects in the product that were not caught during the product development or testing phases.

Automated Software Testing ROI

As we have discussed, metrics help define the progress, health, and quality of an automated testing effort. Without such metrics, it would be practically impossible to quantify, explain with certainty, or demonstrate quality. Along with quality, metrics also help with demonstrating ROI, covered in detail in IAST of this book. ROI measurement, like most metrics, is an ongoing exercise and needs to be closely maintained. Consider the ROI and the various testing metrics

when investigating the quality and value of AST. As shown in Figure 1.7, metrics can assist in presenting the ROI for your effort. Be sure to include all facets in your ROI metric as described in IAST.

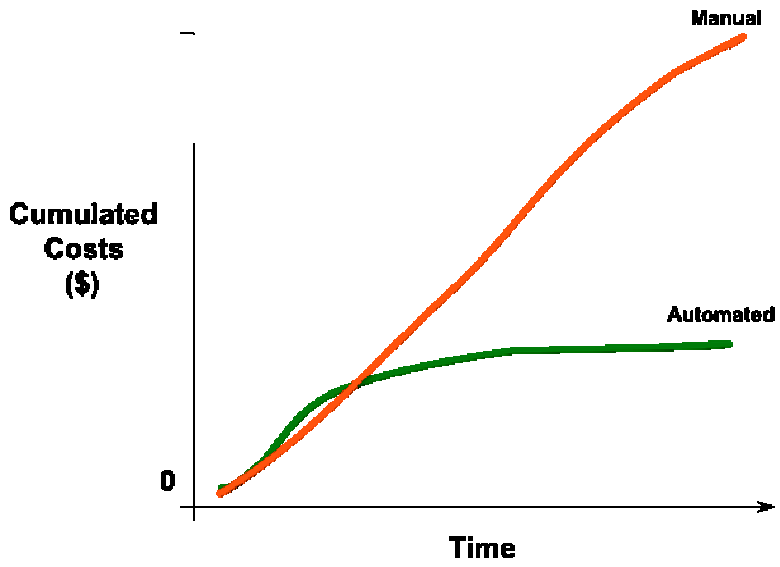


Figure 1.7 AST ROI example (cumulated costs over time)

Other Software Testing Metrics

Along with the metrics mentioned in the previous sections, there are a few more common test metrics useful for the overall testing program. Table 1.1 provides a summary and high-level description of some of these additional useful metrics.

Table 1.1 Additional Common and Useful Software Test Metrics [7]

Metric Name	Description	Category
Error discovery rate	Number of total defects found/Number of test procedures executed The *error discovery rate* is used to analyze and support a rational product release decision.	Progress
Defect aging	The date a defect was opened versus the date the defect was fixed The *defect aging* metric indicates turnaround of the defect.	Progress
Defect fix retest	The date a defect was fixed and released in a new build versus the date the defect was retested The *defect fix retest* metric indicates whether the testing team is retesting the fixes fast enough, in order to get an accurate progress metric.	Progress
Current quality ratio	Number of test procedures successfully executed (without defects) versus the number of test procedures. The *current quality ratio* metric indicates the amount of functionality that has successfully been demonstrated.	Quality
Problem reports by priority	The number of software problem reports broken down by priority The *problem reports* metric counts the number of software problems reported, listed by priority.	Quality

Root Cause Analysis

It is not good enough to conduct lessons learned after the AST program has been implemented. Instead, as soon as a problem is uncovered, regardless of the phase or the type of issue - whether it's a schedule, budget, or software defect problem - a root cause analysis should be conducted, so that corrective actions and adjustments can be made. Root cause analysis should not focus on determining blame; it is a neutral investigation that determines the cause of the problem. For example, a root cause template could be developed where stakeholders can fill in their respective parts if a defect is uncovered in production. The template could list neutral questions, such as, "What is the exact problem and its effect?" "How was it uncovered and who reported it?" "When was it reported?" "Who is/was affected by this problem?" "What is the priority of the problem?" Once all this information has been gathered, stakeholders need to be involved in the discussion and determination of the root cause, how to resolve the issue (corrective action to be taken) and understand the priority of solving and retesting it, and how to prevent that sort of issue from happening in the future.

Defects will be uncovered despite the best-laid plans and implementations; corrective actions, and adjustments are always needed, i.e. expect the unexpected, but have a plan for addressing it Effective AST processes should allow for and support the implementation of necessary corrective actions. They should allow for strategic course correction, schedule adjustments, and deviation from AST phases to adjust to specific project needs, to support continuous process improvement and an ultimate successful delivery.

Root cause analysis is a popular area that has been researched and written about a great deal. What we present here is our approach to implementing it. For more information on root cause analysis and a sample template, review the SixSigma discussion on "Final Solution via Root Cause Analysis (with a Template)." [8]

Summary

To assure AST program success, the AST goals need to be not only defined but also constantly tracked. Defect prevention, AST and other software testing metrics, and root causes analysis implementation are important steps to help prevent, detect, and solve process issues and SUT defects. With the help of these steps the health, quality, and progress of an AST effort can be tracked. These activities can also be used to evaluate past performance, current status, and future trends. Good metrics are objective, measurable, meaningful, and simple, and they have easily obtainable data. Traditional software testing metrics used in software quality engineering can be applied and adapted to AST programs. Some metrics specific to automated testing are

- Percent automatable
- Automation progress
- Percent of automated testing coverage
- Software automation ROI (see IAST for more details)
- Automated test effectiveness (related to ROI)

Evaluate the metrics outcome and adjust accordingly.

Track budgets, schedules, and all AST program-related activities to ensure that your plans will be implemented successfully. Take advantage of peer reviews and inspections, activities that have been proven useful in defect prevention.

As covered in IAST, in the test case requirements-gathering phase of your automation effort, evaluate whether it makes sense to automate or not. Given the set of automatable test cases, determine which ones would provide the biggest ROI. Consider that just because a test is automatable doesn't necessary mean it should be automated. Using this strategy for determining what to automate, you are well on your way to AST success.

References:

1. E. Dustin, "The People Problem," www.stpmag.com/issues/stp-2006-04.pdf; "The Schedule Problem," www.stpmag.com/issues/stp-2006-05.pdf; and "The Processes and Budget Problem," www.stpmag.com/issues/stp-2006-06.pdf.
2. www.thefreedictionary.com/metric.
3. Adapted from Dustin et al., *Automated Software Testing*
4. Stephen H. Kan, *Metrics and Models in Software Quality Engineering, 2nd ed.* (Addison-Wesley, 2003).
5. Adapted from www.teknologika.com/blog/SoftwareDevelopmentMetricsDefectTracking.aspx.
6. C. Jones, keynote address, Fifth International Conference on Software Quality, Austin, TX, 1995.
7. Adapted from Dustin et al., *Automated Software Testing*.
8. www.isixsigma.com/library/content/c050516a.asp

The Spring Framework

Nico Mommaerts, nico.mommaerts@hansei.be
Pieter Degreauwe, pieter.degraeuwe@systemworks.be, http://www.systemworks.be/

Introduction

Before the Spring revolution, enterprise applications were generally written using the J2EE standards (specified by a group of vendors via the JCP process). The premise of J2EE was multiplatform/multivendor, if you would code against the J2EE standards you could deploy your application on any J2EE application server, on any platform. In theory at least. Running your application on an application server has several benefits, the application server offers you services, such as transaction management, messaging, mailing, a directory interface etc. Any J2EE compliant code can make use of these services, as long as the code is written against the interfaces defined in the J2EE specifications.

Unfortunately, there were a few problems with these standards. First of all, the usage of these standards was **too complex**. Writing a component (EJB: Enterprise Java Bean) required you to write a set of xml files (deployment descriptors), home interfaces, remote/local interfaces, etc. Even worse, 50% of the deployment descriptors were vendor specific, so 'transparently migrating an application from vendor A to B' was suddenly not so transparent anymore...

Secondly, there was the 'look-up' problem. When a component requires another component, the components itself was responsible for looking up the components it depends upon. Unfortunately, this look-up happens by name, so the name of the dependency was hardcoded in your component (code or deployment descriptor). On top of that, letting components communicate with each other from J2EE application servers of different vendors was almost always problematic.

Last but not least, in a lot of cases, some components did not need all the services the application server provided, but since there was no other API to build components, all your components became **heavy weight, bloating your application.**

With '*heavy weight*' components we mean that they supported all features (clustering, remoting, etc.) even when we didn't need them. You could say 'but we get it for free, what's the problem?' The problem with this is that 'get it for free' means we do not need to write this functionality, but we need to *configure* them. And this is why most developers longed for writing plain old Java objects, without the need to write ton's of xml configuration for stuff they didn't need...

The POJO Revolution

As said before, one of the goals of these J2EE standards was to give the developer some services for free so he could focus on writing the business logic. This is still a very interesting goal, but it turned out that the programming model was not so flexible as one would like. Coding against the J2EE standards was very cumbersome, you needed to comply to several seemingly arbitrary rules, and it forced you to write code that was not so Object Oriented as one would want to. If you wanted to take advantage of the J2EE services you had to implement your classes in a very specific way, which coupled your business logic to the J2EE classes. It made your code less extensible, and not very test friendly. Since your code used specific J2EE classes and interfaces, you couldn't run that code in a unit test, since it presumes all kinds of services present, normally provided by the J2EE application server. This is what we mean with code that is not testable outside of the container (the application server is the container). Deploying a

J2EE application in a container and starting it up can be a very time-consuming task, so not very ideal if you want to run your unit tests regularly.

More and more developers wanted to write just Plain Old Java Objects (we'll call them POJO's from now on), without the J2EE standards overhead. And to be honest, they were right. Writing business logic in POJO's is way easier. You don't have to implement any interfaces or extend from other classes, you are free to cleanly implement and design your domain with regular Java classes and you don't have to jump through hoops to test your code. The only problem with POJO's is that with J2EE (as it existed then) you cannot benefit from the services provided by the container, such as transaction management, remoting, etc.

This is where the Spring Framework comes in. This framework brings a **lightweight container.** This is a container where your POJO's live in. The difference with a **heavyweight container** is that the components now are as light as possible. You don't have to support services you don't need, this also means no unneeded configuration. You don't have to adhere to some programming model, you can just use POJO's and declaratively specify what services you want to use.

Due to the openness of this container and the use of AOP (Aspect Oriented Programming), it is possible to enhance POJO's in such a way that it doesn't affect the code of your POJO. So, your POJO stays as clean as possible and you can add features as you need them.

The Spring framework, what is it?

It is not a web framework, or a persistence framework, it's a framework that integrates all kinds of Java technologies/API's en makes it possible to use them with simple POJO's.

What is important to know is that Spring does **not reinvent the wheel**. It provides a nice and elegant way to use existing technologies (such as EJB, Hibernate, JDO, Toplink, JMS, etc). This is accomplished by several support classes and 'templates'. Why and how is Spring more elegant? Let's look at an example.

We want to query the number of users that are older than a specified age. Using the conventional methods using the standard JDBC API, we would write something like this:

```
...
  Connection conn = ..;
  PreparedStatement stm;
  ResultSet resultSet;
  try {
    stm = conn.prepareStatement("select count(0) from user u where
u.age > ?");
    stm.setInt(1,age);
    resultSet = stm.executeQuery();
    resultSet.next();
    return resultSet.getIn(1);
  } finally {
    resultSet.close();
    stm.close();
    conn.close();
  }
```

This is already simplified code, since you actually should check for NullPointerExceptions in the finally clause and you should catch SQLExceptions.

However, you can see that this is a very verbose way of writing code. Making use of Spring's JdbcTemplate gives more elegant code:

```
  SimpleJdbcTemplate template = new SimpleJdbcTemplate(datasource);
//this jdbcTemplate supports varArgs.
  int userCount = template.queryForInt("select count(0) from user u
where u.age > ?",age);
```

Of course this is a very simple example, but it should give you an idea how it makes coding Java applications much easier **by removing boilerplate code** from your application code. This example illustrates the JdbcTemplate, however, Spring provides lots of templates: TransactionTemplate, HibernateTemplate, ToplinkTemplate, JDOTemplate, JMSTemplate, etc.

For each supported technology there is a module which consists of helper classes to help you implement a certain layer or aspect of your application. The core of Spring, upon which all other modules depend, is the Inversion of Control and Aspect-Oriented programming module.

It is these two programming models which are the driving force behind Spring. They act like glue, pulling your application together, but in a not-invasive way. Meaning the code you write doesn't need to have Spring references all over the place. In places where you need to use specific frameworks or standards (Hibernate, JMS, etc) Spring lets you integrate easily using these techniques.

What is Inversion of Control?

When developing an application, you always have dependencies between and on components, services, classes etc. Without Inversion of Control you would 'wire' these together on the spot where you would need the dependency. The disadvantage of this is that when you would like to use a different implementation of your dependency, you are forced to change your code. This may not seem that big a disadvantage, but what if you want to change your implementation in context of the environment you are running your code in? For example you might want to use a different authentication service during development as in production. It is not really convenient to change your code every time you need to make a production artifact, or each time you want to run your unit-tests. That's why the wiring of these dependencies is taken out of the code, and an external party manages the wiring, namely the container. Hence the name inversion of control, you let something from the outside control how your dependencies are wired together. We speak of dependency injection because the container 'injects' the necessary dependencies instead of letting the developer manage them.

For example, we have a class PrinterService, which is responsible for sending some document to a Printer. We can have different Printer implementations, laser, inkjet etc.

```
class PrinterService {
  private Printer printer = new InkjetPrinter();

  public PrinterService() {
  }

  public void printDocument(Document doc) {
    printer.print(doc);
  }
}
```

This way, everybody who uses the PrinterService has no choice but to use the InkjetPrinter. Not really what we want, we want to be able to use other printers. If we write a unit test for the PrinterService, we don't want the implementation of the Printer tested, and we certainly don't want a page coming from our printer every time we run the unit test! So we want to be able to change the implementation of the Printer used in the PrinterService.

Let's try the following:

```
class PrinterService {
  private Printer printer;

  public PrinterService() {
  }

  public void setPrinter(Printer printer) {
    this.printer = printer;

  }

  public void printDocument(Document doc) {
    printer.print(doc);

  }

}
```

Now if someone wants to use the PrinterService he can do something like this:

```
PrinterService printerService = new PrinterService();
printerService.setPrinter(new LaserPrinter());

printerService.printDocument(someDocument);
```

As you can see the usage of the PrinterService isn't limited anymore to the InkjetPrinter we defined in the first example. However, we still have to wire the Printer implementation to the PrinterService ourselves. If we want to run this code in a different context, with a different printer implementation, one for testing purposes for example, we still have to branch somewhere in our code to make a distinction between the different contexts/implementations.

Now this is where Spring as a container comes into the picture. We want to let Spring handle the wiring of the dependencies of our PrinterService. To do this we let the PrinterService and the Printer 'live' inside the Spring container, as such they are called beans. We give each bean a unique name, an id, so we can reference them later on. We can declare this in a Spring xml configuration file (xml schema declaration omitted for brevity):

```
<beans>
  <bean id="laserPrinter" class="com.methodsandtools.LaserPrinter"/>
  <bean id="inkjetPrinter" class="com.methodsandtools.InkjetPrinter"/>

  <bean id="printerService"
class="com.methodsandtools.PrinterService">
    <property name="printer" ref="laserPrinter"/>

  </bean>
</beans>
```

In this example we defined two Printer beans, 'laserPrinter' and 'inkjetPrinter'. We used the property 'printer' on the 'printerService' to wire the 'laserPrinter' to the 'printerService' bean. What actually happens is that the 'setPrinter' method on the 'printerService' bean is called with the 'laserPrinter' bean as argument. These beans and their lifecycle are now managed by the Spring container, we could configure if we want a new instance of the laserPrinter bean be created every time we inject it somewhere, or if we only want one instance of the laserPrinter bean to live in the container and that one be used all the time. Managing this lifecycle is beyond the scope of this article, but is nevertheless a very interesting and necessary capability of the Spring container. Now, if we want to use the PrinterService, we have to ask the Spring container the bean we declared in the configuration file. The Spring container is accessed through a bean factory:

```
XmlBeanFactory beanFactory = new XmlBeanFactory(new
ClassPathResource("configuration.xml", getClass()));
PrinterService printerService =
(PrinterService) beanFactory .getBean("printerService");
```

We now have a PrinterService instance which we can use without worrying what Printer implementation is wired to it. In a different context we can use a different configuration file, no code needs to be changed in order to use a different Printer implementation. You can also split the declaration of the beans over multiple xml files, so you can pick and mix to your needs.

Note that the above code doesn't need to run in a J2EE container, meaning you don't need to start an application server and deploy it, it can be run as a regular Java program.

Although we declared the bean factory ourselves in this example, usually in a J2EE application there will be a bean factory associated with a context defined by the J2EE server and you won't see any Spring code in the business logic. So you are not tied to using an API or implement some interfaces as with standard J2EE to use the Inversion of Control container.

Once you start working this way, you'll soon start thinking in terms of declaring beans and wiring them together. There are more possibilities as demonstrated, like using inheritance between beans, controlling the lifecycle, autowiring dependencies based upon type, etc, but the above example should give you a basic understanding of the power of Dependency Injection in Spring.

What is Aspect-Oriented programming?

In most applications there are concerns that 'cut' across different abstraction layers, the typical example is logging. You might want to log in every method of your service layer that you are entering and exiting that specific method. This litters log statements all over your service layer, while logging is actually one concern and as such should be separated from the business logic into a different entity. This is what Aspect-Oriented Programming frameworks (we'll cal it AOP from now) aim to do. In AOP terminology a concern is written as an advice, which is an entity like a class. Then this advice can be applied to certain pointcuts in your code. A pointcut is a place or several places in your code where you want the crosscutting concern, the advice, to be applied.

For example when entering or exiting a certain method. An advice together with a pointcut is called an aspect, hence the name Aspect-Oriented Programming. There are many AOP frameworks out there, Spring uses its own based upon dynamic proxies and/or CGLIB byte code generation, but can integrate with others like AspectJ if desired.

A simple example of an advice, straight from the Spring javadocs:

```
class TracingInterceptor implements MethodInterceptor {
  Object invoke(MethodInvocation i) throws Throwable {
  System.out.println("method " + i.getMethod() + " is called on " +
i.getThis() + " with args " + i.getArguments());
  Object ret = i.proceed();
  System.out.println("method " + i.getMethod() + " returns " + ret);
  return ret;
  }
}
```

The above advice is invoked when a method is intercepted, which methods will be intercepted is defined in the pointcut, more on this later. The 'invoke' method receives a MethodInvocation instance which contains all the needed information about the method being intercepted. This can be seen in the System.out statement where we print all this information. The next line is also very interesting, we tell the method being intercepted to proceed, so this means that if we would like to, we could prevent the invocation of the intercepted method!

Now that we have defined what to do with the intercepted method, we have to define which methods will be intercepted. These two concepts are separated, which is a powerful principle. Saying which methods to advice is done in a pointcut, which is also an interface to be implemented, the Pointcut interface, which looks like this:

```
public interface Pointcut {
  ClassFilter getClassFilter();
  MethodMatcher getMethodMatcher();
}
```

For the most common cases there are a lot of standard implementations of this interface in the Spring framework.

One of the most (in)famous of the J2EE standards is the EJB standard, Enterprise Java Beans. Among other things, it provides declarative transaction management for your database access code. Managing your transactions in your database access code is something you have to do each time you make a call to the database, so being able to do this declaratively in an xml file was a real advantage. However, to benefit from this you had to code against an API, which sometimes meant having up to 7 files for one EJB, having to implement arbitrary methods, and your EJB would only work if deployed in a J2EE application server. This is just one example of the services that J2EE provides, but is a real pain to use because of the API you have to program against. What Spring aims to do is to give you the same services as provided by J2EE, e.g. transaction management, but in such a way that you can still code using POJO's. Spring accomplishes this by using AOP.

Testability

As said before, writing tests for EJB's were quite problematic, since your components (EJB's) had to run in a running container.

Writing your components in the POJO model solves this completely. But that is not all, Spring brings some nice features for writing integration tests. Especially useful are the transactional tests, which you can configure to rollback all inserted data, so your test database stays clean and won't interfere with other tests or the same test if it should fail. You are not required to manually clean up your test data after each test.

Let's look at an example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/applicationContext.xml",
"/database-test.xml" })

@Transactional
public class FictitiousTransactionalTest {

  @Autowired //inject by type
  private PetClinicService petClinicService;

  @Resource(name="hibernateTemplate") //inject by name
  private HibernateTemplate hibernateTemplate;

  @Before
  public void setUpTestDataWithinTransaction() {
    // set up test data within the transaction
  }

  @Test
  public void modifyDatabaseWithinTransaction() {
    // logic which uses the test data and modifies database state
    hibernateTemplate.save(VetTestObjectMother.createDummyVet());
    hibernateTemplate.save(VetTestObjectMother.createDummyVet());

    assertEquals(2, petClinicService.countVet());
  }

  @Test
  @NotTransactional
  public void performNonDatabaseRelatedAction() {
  // logic which does not modify database state
  }
}
```

First of all, we can define the Spring context files (the files where we wire all POJO's together), that should be loaded by the integration test. (@ContextConfiguration)

Secondly, we can inject some components into our test, so, we can start testing them. (@Resource and @Autowired)

For integration tests, it is especially useful that we can perform our tests in one transaction, and rollback that transaction after the test has run. (@Transactional, @NotTransactional, @BeforeTransaction, @AfterTransaction, @TransactionConfiguration)

Conclusion: To Spring or not To Spring?

As you might have already understood, we really think that most applications (if not all) should use a lightweight Dependency Injection container.

Inversion of Control promotes loose coupling and testability of your code, the POJO programming model is how it should have always been. (IoC really enables testabillity of the code, the POJO programming model, etc). Spring is the most popular and widely accepted DI container there is for Java. It's entirely up to you how much you use it. You can choose to just use some of the supporting classes or templates, or you can go full out and make your entire application use Spring beans. Besides the core Spring functionality which we covered here, over the years a lot of extra modules appeared which do replace existing technologies. For example Spring Web Flow is a very nice framework built upon the Spring framework for building Web applications.

With Spring it's your choice which API's you use, and how you build your application.

Reference

The Spring Framework:
http://www.springsource.org/about
http://en.wikipedia.org/wiki/Spring_Framework
http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework

Spring Web Flow:
http://www.springsource.org/webflow
http://www.ervacon.com/products/swf/intro/index.html

J2EE:
http://www.j2ee.me/javaee/
http://en.wikipedia.org/wiki/J2ee

EJB:
http://java.sun.com/products/ejb/docs.html

AOP:
http://en.wikipedia.org/wiki/Aspect-oriented_programming
http://onjava.com/pub/a/onjava/2004/07/14/springaop.html

IOC:
http://en.wikipedia.org/wiki/Inversion_of_Control
http://martinfowler.com/articles/injection.html

Dependency Injection:
http://en.wikipedia.org/wiki/Dependency_injection
http://www.theserverside.com/tt/articles/article.tss?l=IOCBeginners

The Learning View

Allan Kelly, allan@allankelly.net
www.allankelly.net

That software development entails a lot of learning should not be a surprising revelation, but in fact everything done in developing software is a form of learning. And when the development team are finished with the software and it is opened up to the final users they too will need to learn and to change the way they work.

Once the learning view is recognised it can be harnessed to help improve all aspects of software development: requirements, coding, testing and delivery. For the teams that learn fastest, and put their learning into action there are significant rewards.

Learning all around

Learning occurs in three domains. First is the solution domain: that is the tools and technologies used to create software products. The constant stream of new languages, new API's, revised operating systems, databases and other technology presents the most obvious need to learn.

Next is the application domain – the business area and problem that is being solved. Software coders and testers who spend any amount of time working in the same business area end up being minor domain experts in their own right. I once worked for in the forecasting department at an electricity company where I picked up enough knowledge of electricity markets to become a minor expert myself.

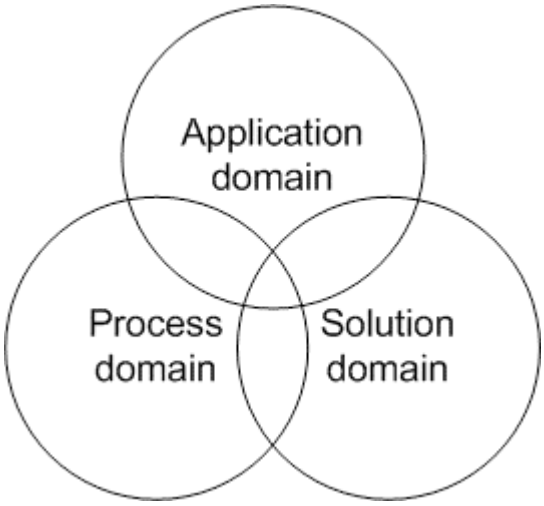


Figure 1 - Three domains of learning

The creation of software occurs at the intersection of these two domains, where the tools are brought into contact with the problem to be solved. The problem solving that occurs here is itself a learning activity: coders and testers learn what works and what doesn't, they learn what the customers really wanted as opposed to what they thought they wanted.

Finally there is one more domain where learning occurs: the process domain. How development efforts are organised: whether work is run Agile or Waterfall, whether Testers are embedded with Coders or separate, how requirements are communicated, and much more.

True learning involves change

In all domains true learning implies change. If I learn a fact but do not act on it then nothing has changed and no learning has really occurred. However when I act on what I know a deeper learning occurs and knowledge is created.

For example, if I "learn" Ruby on Rails but never use it to write code then I may know the syntax but nothing has changed, the learning is of little value. Yet if I use my new learning to develop a new website, or rewrite part of an existing system then a deeper learning has happened and change has occurred.

This phenomenon is most apparent in the process domain: reading a book on Scrum does not make a difference unless one puts this information into action. Similarly, holding a project retrospective and not acting on the insights adds nothing to our stock of knowledge.

So while it is important to learn one must also put that learning into action. Action completes the learning cycle. The problem solving work that developers and testers engage in to write and test code satisfies this criteria: developers think a bit, they code a bit, they learn a bit and this in turn feeds another iteration around the same loop. Likewise for testers, working with an application seeds the learning loop.

With learning taking centre stage in the development process it follows that those who can learn fastest stand to benefit most.

Learning into action

While traditional training courses are good at educating people they are not so good at creating change. As at school, the nature of training tends to be passive: students sit and listen to an expert. This is a good way of communicating information but does not impact any sense of action.

One result of this approach can be individuals who "can speak the language" but do not act on it. Consequently the passive learning process can actually reinforce the status quo. The situation is compounded when people receive training weeks or even months before they are expected to us it.

In order to create change individuals need to be more involved in the learning process and actively experience something new. Good training courses use exercises, discussion and problem solving to challenge students' ideas and experience new material.

Retrospectives

Perhaps the most well known technique for promoting learning and change in software development teams is the retrospective. Theses sessions allow team members to look back on recent work, reflect on what was done and suggest improvements for future work. The key to a good retrospective is ensuring that change happens after the retrospective.

While a retrospective may generate dozens of suggestions for improvement and change if none are acted on then the retrospective is of little value. One way to ensure action occurs is to focus on a small number of changes, say two or three, and ensure these suggestions are put into action. Other suggestions may be held over and considered in a future retrospective.

Future-specitves

Although retrospectives are normally held after a development episode it is possible to use similar techniques and approach at the start of work. I sometimes refer to these sessions as *future-spectives*.

A future-spective could be seeded by saying something like: "Imagine we have come to the end of the first release. What have we done right?" Or, you could say: "What are the things we think we should be doing?"

After priming the future-spective continues by brainstorming. People suggest ideas either by writing them on index cards, or by verbally suggesting the idea and a facilitator records the suggestion on a flip-chart or similar. After a good body of ideas has been collected the brain storming closes and the ideas are reviewed.

Reviewing the ideas usually starts with talking about how the idea would work in practice. This is the point when ideas can be questioned and sometimes withdrawn. Typically there is some duplication and ideas are merged.

As with retrospectives there is usually no shortage of ideas, the problem is getting the ideas into practice. Rather than try and do 20 things differently it is usually better to focus on a few ideas. To ensure buy-in from the team some sort of voting system, such as a dot-vote, can be used to select the ideas to be changed.

Don't worry about the other items you don't choose right now, you have a few options. Either keep the list and review it again in a few weeks; or hold another future-spective when the first changes are in effect; or switch to a retrospective format in future.

Once you've got your few suggestions spend more time with the group talking about what is involved and who will need to do what. I used this format with one group a few years ago. The top suggestion, by a long way, was to start doing Test Driven Development. When we talked about it in more detail it turned out it meant different things to different people.

Wherever possible you need to get people to talk about what will constitute success with the suggestion. Ask questions like "What does it look like when we do that?" and "How will we measure our success?"

Different suggestions will need to be implemented in different ways. Some will be as simply as adding a new work item to the task list, say, "Rebuild test environment" while others may be more complicated and require a different approach. Of course some ideas are bigger than others. A small suggestion like "Start holding morning stand-up meetings" is far easier to do than "Adopt Scrum as out process."

The future-spective approach has the advantage that it doesn't dwell on all the problems, conflict and any negativity that may have built up in the past. There is a school of thought – called *Appreciative Inquiry* – which holds that examining problems invites negativity in. Rather than dwelling on what went wrong it is better to talk about what went right. Rather then discussing past problems it is better to talk about a bright, successfully, future.

I find future-spectives a useful tool when I first start working with a team. As a coach or an interim manager I am tasked with changing things. Holding a future-spective draws a line under the past without examining it in too much detail and starts imagining a better future.

Personal journal

At an individual level keeping a personal journal, or reflective diary, can be a useful way to reflect on the work and events. I try to write my journal at least once a week, what I write is largely a stream of consciousness to start with. Sometimes it is a list of things that have happened recently; sometimes it is a situation statement on problems I face or recent achievements. After a few sentences I usually start to get insights and ideas for tackling the upcoming work.

A journal is a way of helping yourself make sense of the world that is around you. In so doing it is an opportunity reflect on what can be done to improve matters. By writing you slow your thought process down and put it in order.

Blogging can fill a similar role but the privacy of a personal journal can be useful when considering details and people. A private journal avoids the risk that someone on your team will take things the wrong way so you can write more freely.

The journal is also an opportunity to consider what might happen, and what action could be taken if the right occasion arose. As Louis Pasteur is reported to have said “Chance favours the prepared mind.” By writing a journal you can gain insights which allow you to seize opportunities when they occur.

There are no hard and fast rules for how you go about writing a journal or structuring your thoughts. Nor do you need any special tools, you can buy journal software but a word processor will do find. Indeed I imagine some people will prefer to use pen and paper.

I find it best to keep the journal regularly, it can be hard to find time in a busy work environment so you might want to block some time out or, if you can work from home occasionally, do it there.

While you might include a list of events that have happened this should only be to help you analyse and think about things. Try to keep the journal focused on the future and directed towards improvement.

Bring ideas inside

While retrospectives and journals can be useful for understanding our work and deciding on causes of action it also helps to open minds to new ideas and approaches. Reading books, magazines, mailing lists and blogs can help here but for high bandwidth communication little beats talk.

Stepping outside a company to attend conferences or evening talk sessions is a useful way of exposing yourself to new ideas. The conversations following presentations can often be as enlightening as the formal presentation itself.

Inside the company regular talk programmes can help both spread new ideas and share information. Running these over lunch times - with a little food– can provide a more relaxed atmosphere and be an effective use of people’s time.

The key to such sessions is regularity. All companies have internal talks from time to time but when a regular session exists it becomes part of the fabric of the company. At first you might want to focus on technical issues – have a team present their latest work, or a developer

demonstrate a new language. In time you may want to get the marketing department to talk about their campaigns, or have a product manager talk about the competition. And of course, if someone goes to a conference they could present a report on what they learned.

Running a talk programme – often called *TechTalks* or *Lunch & Learn* – can be a challenge. Finding the speakers, publicising the event, including remote offices and keeping the schedule together all take time.

A Wiki can help organise the sessions and make the schedule public; still it is worth sending e-mails to remind people. Some companies task an administrative assistant to help run the programme.

In a small company it might be best to schedule talks every other week. Don't be disheartened when only 3-4 people turn up to a session, it happens. Keep you ears open for interesting ideas and ask people what they would like to hear about.

Most likely you will normally use a lecture style format but be prepared to try new formats. Interactive sessions, demonstrations and discussions are all good. Think about inviting guest speakers in to liven up the programme, introduce new ideas and excite interest. Some guests will speak for free while others might expect a payment or at least expenses.

Conclusion

Learning is not something that just happens in a classroom or when you read a book. In fact very little real learning occurs in either case. Those working in software development are constantly learning, creating knowledge and putting that knowledge to work in everyday work.
It is because learning is constantly occurring that it is not possible to write complete specifications or know exactly when a piece of work will be finished. Rather than see this as an engineering problem to be solved it should be seen as an opportunity. Improving the ability of a team to learn, and helping them change, will create more effective teams.

Finally, learning occurs however software is developed and these techniques can be used with any technology, methodology or process; but they are most applicable in an Agile context. This is simply because Agile is rooted in the adaptation and change, in other words, learning.

Time and Synchronization in Executable UML

Leon Starr leon_starr@modelint.com
Model Integration, LLC, www.modelint.com

Abstract

This article illustrates the platform independent timing and synchronization rules that modelers use and architects implement to support Executable UML® (UML is a registered trademark of the Object Management Group) applications. UML standardizes behavioral notations such as statecharts and sequence diagrams, but it does not define any synchronization rules. One of the most powerful features of Executable UML is that it does not constrain the implementation with unnecessary sequencing. In traditional programming languages, the stream of processing is sequential by default unless intentionally diverted through a variety of platform specific concurrency mechanisms. The opposite is true in Executable UML. Everything happens in a concurrent, platform independent timeframe unless there is explicit synchronization. This allows us to create application models that can be deployed onto arbitrarily distributed platforms. This capability is increasingly relevant as platform technology evolves from embedded to parallel to distributed to cloud and back, sometimes in the duration of a single project! So it is ever more critical that all the hard work you put into your analysis, models and designs not fall apart as the target platform inevitably shifts.

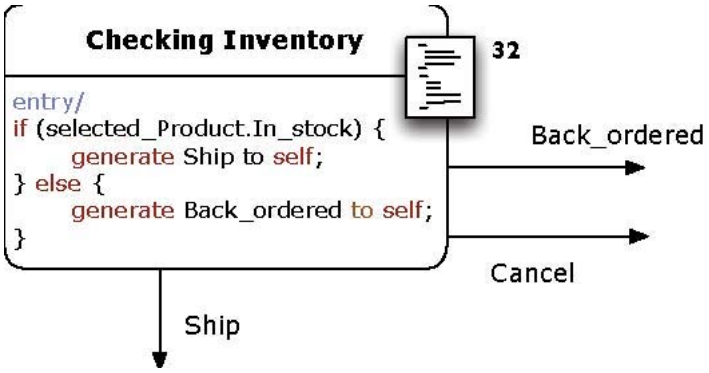
UML does not Prescribe Synchronization Rules

With all of the dynamic diagram types in UML: sequence diagrams, collaboration diagrams, statecharts, activity diagrams and such, it may be surprising to learn that UML does not supply any timing and synchronization rules. This is simultaneously a feature and a curse. It is a feature in that you are free to define your own rules. It is a curse in that... you are free to define your own rules.

The notational limits of statecharts when it comes to definitive synchronization can be illustrated with a few simple examples.

Ambiguous example 1: Busy when event occurs

In this order processing application we have an object, Order 32 sitting in the CHECKING INVENTORY state. It is executing the procedure inside the state when a Cancel event occurs.

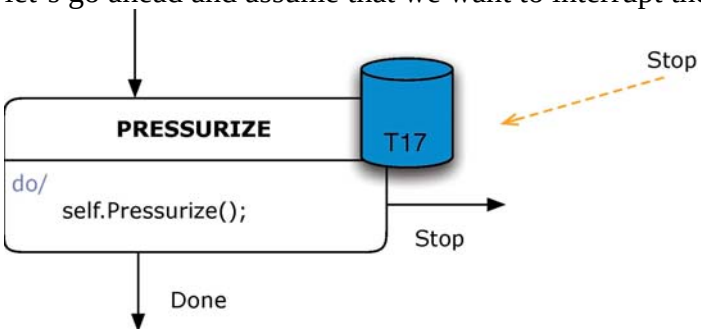


What does Order 32 do when a Cancel event occurs during processing?

Is the processing interrupted and the Cancel transition taken? If so, what about any cleanup required? There is probably none in this particular case, but you can imagine what might be required in a more complex procedure. Or does the event get dropped since it arrives while the Order object is busy? Or could the event be saved until completion of the procedure? The point here is not that any particular rule is best, just that you need rules. UML can accommodate a range of possibilities, including none. Let's take a look at another example.

Ambiguous example 2: Interrupting a continuous action

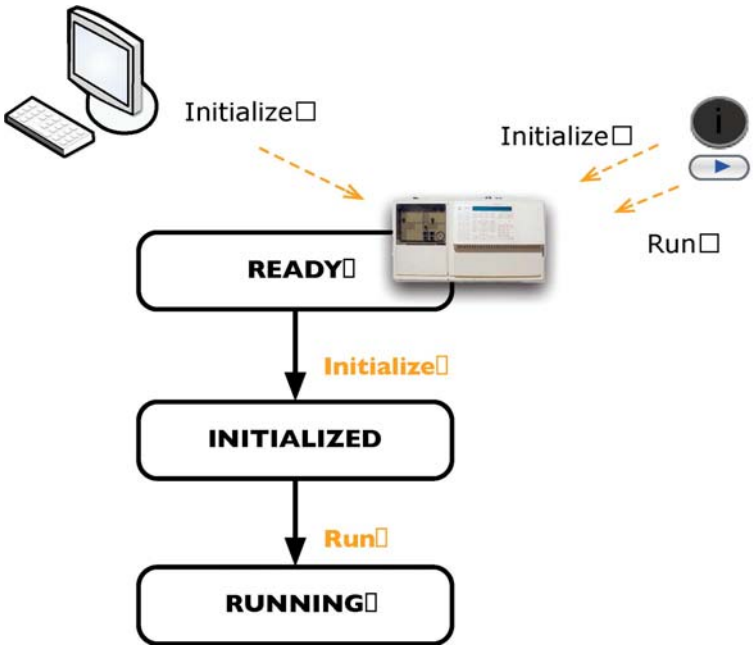
The mixing tank object T17 is executing a **do/** action when the Reset event occurs. In this case, let's go ahead and assume that we want to interrupt the activity.



How, exactly, will this be handled? At what level of granularity is interruption possible? And, again, what about cleanup? The statechart offers notation, but no guidance as to how to process the interrupt. Okay, one last example...

Ambiguous example 3: Sequencing events

Here we have a chemical measurement instrument commanded from buttons on the front panel of the device and/or via a remote workstation connected on a network. A human operator first presses the Initialize and then the Run buttons triggering the corresponding events. Simultaneously, a command that triggers another Initialize event arrives on the network. (Perhaps the operator and remote user are unaware of the potential conflict). What happens?



Are the two events commanded from the button panel processed in the intended order? Could the network Initialize event happen in between, and if so, will it be ignored or processed?

At the end of this article we will revisit these three examples and see how Executable UML resolves the ambiguities.

Resolving Synchronization Ambiguity

In these three examples we've seen that UML notation alone does not resolve real-time ambiguity in statecharts. Given that we are modeling a real-time system, what are the options for proceeding?

Option 1 - Informal specification

Maybe you are building high level models that aren't intended to solve any synchronization problems. You just want a very informal specification of behavior and plan to leave all the hard work to the developers. Leave it to the coders and fix things in beta. Yee-ha!

Option 2 - Mix code into the specification

We can elaborate the weak statecharts by writing the procedure in each state using a traditional programming language like C/C++ or Java. The code can invoke concurrent operating system facilities such as threads, semaphores, critical section marks and so forth. This may work, but it doesn't guarantee consistent rules across all statecharts (or even within a single statechart), so the models may be quite complex in the end. Let's assume though, that considerable thought goes into the coding and a relatively simple and global scheme is devised to keep the

complexity from getting out of hand. You will then end up with largely platform specific models. This compromises the quality of the analysis (mixed in implementation complexity), precludes early simulation and testing (you've got to write all the code first) and limits the models to the target platform (redesign and retest required to move to another platform).

So the procedure coding approach may address the problem of implementing synchronization, but you have to tread through a lot of implementation muck to get there. (And you have to keep treading each time you build a statechart!) The difficulties of debugging systems that use concurrency mechanisms are well documented out on the Internet.

Option 3 - Model synchronization requirements with UML

Here, the idea is to apply a simple set of rules consistently across all of the statecharts. Both the analyst and the designer benefit from this scheme. The analyst need not get bogged down in complex synchronization mechanics and can build models that clearly expose [1] application requirements. The designer can more easily implement a lean, efficient state machine engine based on a uniform rule kernel. The designer need not pick through and accommodate varied and specialized control mechanisms scattered and buried inside the actions of each statechart.

The synchronization rules may be simple, but they must simultaneously offer the ability to capture complex application requirements and be implementable on a wide variety of platform technologies.

Consider some of the advantages over options 1 and 2 if this can be done.

You will be able to build unambiguous, testable models of complex application behavior. These models will be platform independent (PIMs). [2] In other words, the implementation specific concurrency and synchronization mechanisms will be factored out of the application models. What will remain are the application specific synchronization requirements essential to all implementations. Well defined execution rules make it possible to run the models on a virtual machine for testing purposes prior to implementation. Rough metrics concerning peak and average signal/data exchange, relative action execution and dwell time and various other latencies can be gathered. Finally, a single executable domain can be implemented on a variety of concurrent platforms without necessitating changes to the models themselves. A domain is a type of package that contains a complete set of executable content, modeled or coded. There are some other Executable UML specific definitions of a domain described in Mellor-Balcer[3]. The accompanying marking model (MDA terminology [2]), however, will likely change. It marks various model features to be handled specially by the model compiler. Each MVM will make various marking features available appropriate to the class of targeted platform. For example, certain classes might be marked for flash memory implementation. This means that once the application synchronization is tested and validated it need not be reworked and retested just because a platform with varying distribution, timing and synchronization mechanisms is targeted.

These advantages should be compelling for anyone building complex real-time systems that must accommodate evolving platform technology. (Especially if task or processor distribution is in flux)

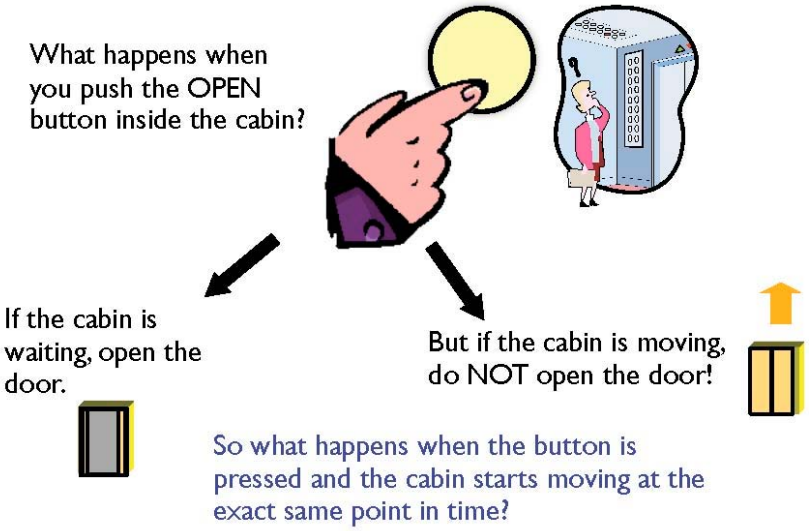
In this article we will explore option 3 as plenty has already been written about options 1 and 2!

But first, a lot of assumptions have been piling up, not least of which is that it is even possible for synchronization to be platform independent. Let's see.

An Example of Platform Independent Timing and Synchronization

Some developers believe that timing and synchronization can be sorted out only with the platform and target programming language taken fully taken into consideration. Without a detailed understanding of how threads, tasks, semaphores and other synchronization facilities, including the hardware, work, how can you definitively express and test any synchronization scheme? This is the thinking pushing most developers toward option 2. But maybe it doesn't have to be that way.

Consider a synchronization problem in an elevator control system. That, of course, would be a "lift" control system for my British friends A passenger standing inside the cabin is going to push the OPEN button. If the cabin is at rest, the doors should open. If the doors are already open, nothing happens. If the cabin is moving and the doors are closed, again, nothing should happen. Now what happens if the open button is pressed at the exact same instant that the cabin wants to move?



Is it necessary to consult the pthreads reference book before you can answer? Maybe dust off your favorite real-time OS manual just to be sure? Of course not! You already know what is supposed to happen. The cabin can never be moving while the doors are open. But how do we express the resolution of simultaneous events? Informal text can't solve the problem. An implementation certainly can.

Option 3 says that we need to find a middle ground where we can unambiguously model the synchronization requirement in a single Executable UML model. This model can then be transformed into many different code sets, each implementing the exact same rules but using the unique features of a particular platform. How do we do this?

Defining and executing platform independent synchronization rules

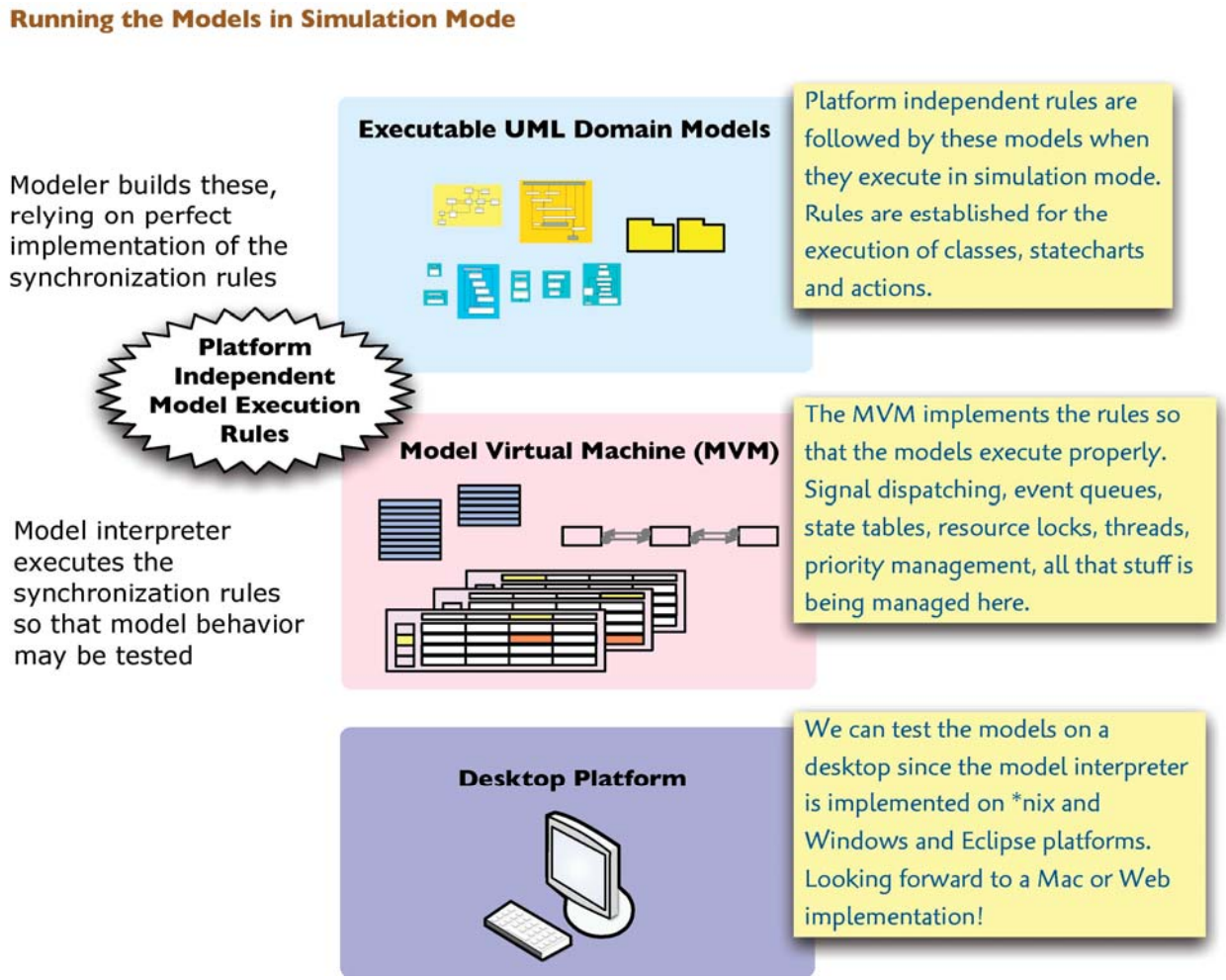
First we define a set of rules for interpreting statechart behavior. Actually, we don't have to define them since they are conveniently defined in Executable UML. (Though there is nothing to stop[6] you from defining your own ! Well, ... time and money, perhaps. More about this at the end of the article.) For example, one Executable UML rule says that "a procedure in a state always completes before the object processes any new events". We'll get to the rest of these rules soon.

A modeler / analyst will then model one or more application domains using class, state and action models that conform to the Executable UML rules.

During development, the statecharts may populated with instance data and executed either by hand (coins on sheets of paper) or, ideally, using a model interpreter that supports the Executable UML rules. These do exist, or you can build your own if you are looking for something to do.

The model interpreter or model virtual machine (MVM) must be constructed so that the rules observed by the modeler are correctly implemented.

Here is an environment that executes UML models on a desktop:



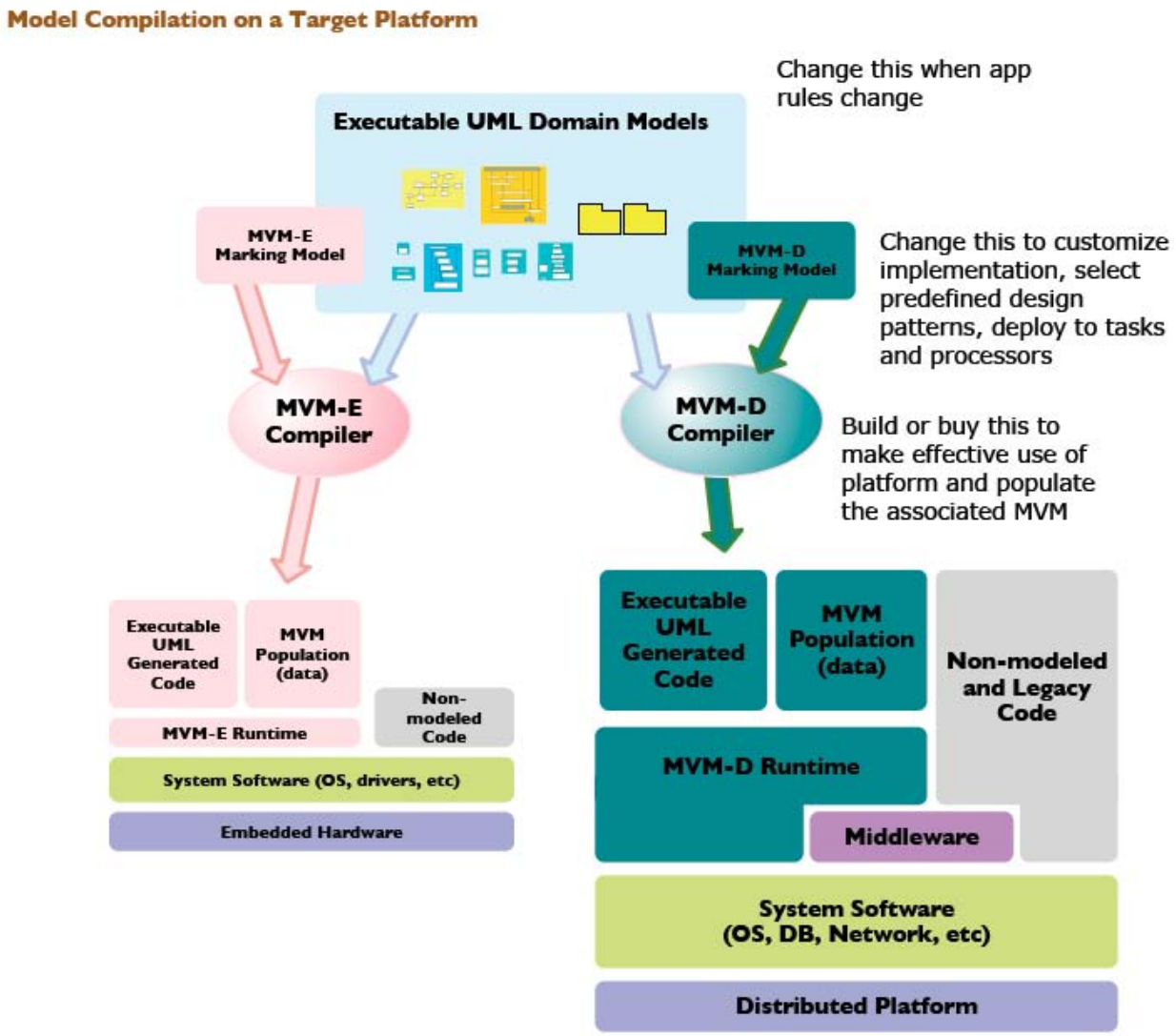
The MVM will allow us to put all of our objects into an initial state, trigger an initial event and then let the models execute actions and move state by state. Ideally, breakpoints can be set on states, events and individual actions within procedures. (It's actually more than an ideal. These things do exist out in the wild). There is, of course, more to the MVM than just executing state synchronization rules. Some UML actions access attributes and navigate class model relationships. So, in addition to statechart rules, executable data and computation semantics must also be supported by the MVM

Simulation is Fine, but what about Implementation on the Target Platform?

It's nice to run your models and test them against platform independent synchronization rules. But the desktop implementation of the model execution rules will be inappropriate for a highly embedded target or a highly distributed target. In fact, the desktop model interpreter is just one of many possible platform targets.

Let's say that we are compiling to a highly embedded platform. In that case you are probably implementing in C instead of Java. Yes, I know there's an embedded Java runtime environment, but I'm talking highly embedded, 50 - 256k, for example. There are many places where a Java virtual machine won't fit. But you can find room for the model execution rules, though, in extreme environments, you will limit the types of applications that can be supported.

So the implementation solution is not to require that each platform accommodate a full scale Executable UML virtual machine. Instead, application realities such as the quantities of persistent and non-persistent objects, quantity of read only data and lots of other factors can be taken into account to create an Executable UML run time environment suited to a class of platform.



Guiding Principles of the Time and Synchronization Rules

Having surveyed the benefits desired and the technical concerns that must be addressed, we can list the guiding principles of the Executable UML time and synchronization rules:

1) Simplicity

2) Minimize platform assumptions

3) Leverage existing theory

Simplicity

Define as few rules as necessary. There's enough complexity in the application without folding in a set of synchronization mechanics riddled with exceptions, redundancies, event juggling and priority schemes. As stated earlier, this benefits both the analyst and the designer. Subtle bugs often result because faults in the application logic are clouded by complex language mechanics. Executable UML models are quite easy to review compared to a lot of the UML gunk out there.

Minimize Platform Assumptions

If the synchronization, data access and other execution rules are truly platform independent, the application models will be deployable on a variety of platforms. Since the platform is not known ahead of time, it is best to assume the worst. A rule assuming a global clock is available, for example, would limit deployment to single CPU targets. An assumption that a group of classes in a subsystem should run in the same thread will have limited portability. So we would prefer to go the other way and make the most minimal assumptions about synchronization possible knowing that the models, once tested, will work the same way on any arbitrarily distributed platform.

Leverage Existing Theory

Executable UML builds on existing theory. The data semantics, for example, are built on relational theory. The synchronization rules borrow from distributed networking theory [4].

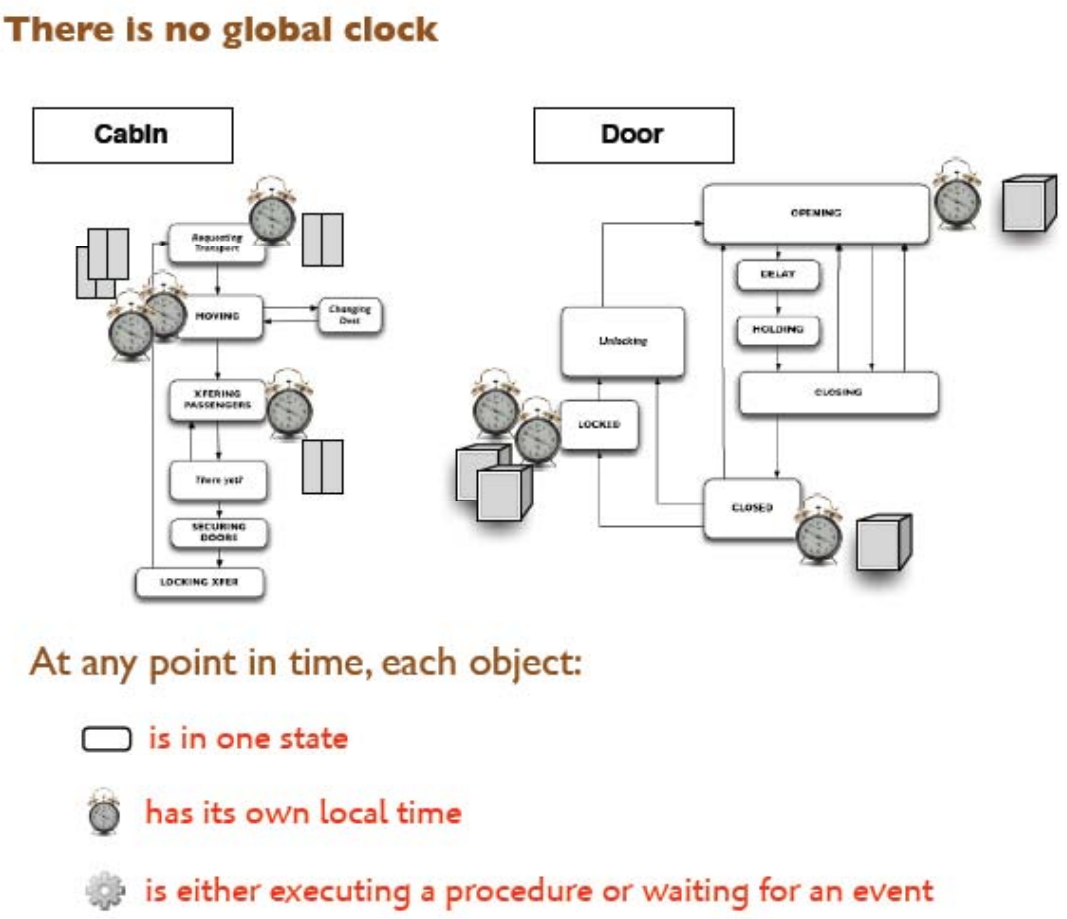
Now, on to the rules.

The Rules

There is no Global Clock

The concept of a global clock, where all model elements are running in the same absolute timeframe is difficult and often impossible to implement on distributed or multi-processor platforms. It is a nice convenience if you are targeting a single CPU, but we can't count on that. In fact, the actual implementation of the object state machines could be looping in a single thread within a single task in the degenerate case of distribution. At the other extreme, each object could be running in its own processor. By assuming the worst case of distribution, we know that we can target all cases.

So there is no global clock. We assume that time is local to each object. In this illustration we depict each object running in its own local time.



Each class may be populated with some number of objects. Here we are concerned only with those classes with statecharts. I like to picture the statechart of a class like a game board traversed by each object of the class. At any frozen point in time, each object is in exactly one state of its class's statechart.

So all objects belonging to the same class are driven by a single statechart. Across multiple classes, we have all of the objects, each in a current state either executing the entry procedure within that state or just waiting around for an event to occur.

Obviously there are a few implications to this way of thinking that must be taken into account by the remaining rules.

THE DURATION OF A PROCEDURE IS UNKNOWN

Each state contains a single entry procedure that we will refer to as the state's "procedure" from here on as done in Mellor-Balcer[3]. Within this block there may be multiple actions. It is also possible for the entry procedure to be empty or to just contain a comment.

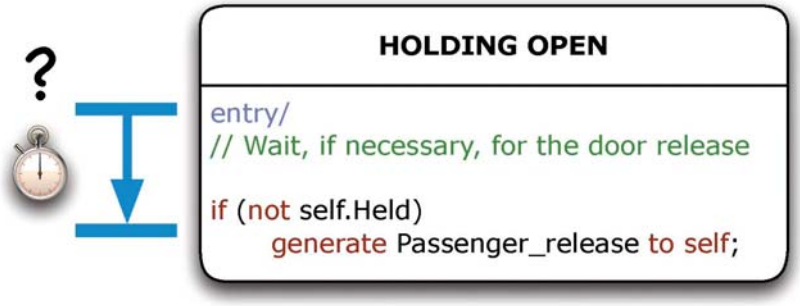
While standard UML notation provides for entry, exit and do actions, Executable UML uses only entry actions. Why are the other two omitted? Simplicity. It turns out that you can do it all with entry actions. You can also do it all with exit actions, for that matter. In the development of the Executable UML profile, much attention has been focused on having one way to solve a problem instead of fifteen. That way a given set of requirements should reduce down to a common model rather than multiple alternatives with no objective criteria for distinguishing one from the next.

It is a lot easier to read through a collection of statecharts all using entry actions than it is to parse through a different scenario for every state.

You may also note that procedures are associated with states (Moore style) as opposed to transitions (Mealy style). The debate as to which style is better is akin to that for curly braces in C code and just as tedious. There is no significant complexity difference between styles. The Moore style used in Executable UML does lead to a simplified state table (the thing that actually gets compiled) and makes it easy to specify the re-execution of an action (go back to the same state).

When an object enters a state, it begins executing the entire procedure immediately. At this point in time we will call it a busy object. It takes an unknown period of time to complete the procedure. That is because the speed of execution is platform dependent. Upon completion, the object waits for an event to occur. Objects never automatically advance to another state. At this point in time we have a waiting object.

Platform dependent duration



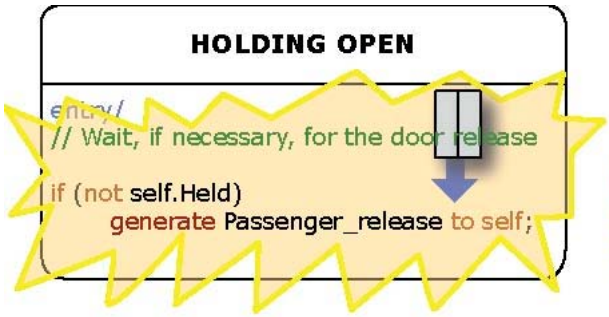
In a platform independent world, the duration of procedure execution is finite, but unknown.

Events are Held for Busy Objects

Waiting objects are idle until one or more events occur. If one of these events corresponds to a transition exiting the current state, the waiting object will follow the transition and get busy again.

Busy objects do not *see* any events. When events targeted at a busy object occur, they are held. At least one of these held events will be become visible when the busy object has completed its procedure and becomes a waiting object.

Busy Object



If events occur now, they will not be seen by the busy object. But they will be held by the MVM for later processing.

Waiting Object

After the action is executed, the object can see pending or newly occurring events.

The object waits in this state until an event occurs that triggers a transition.

Implications for the modeler

Since you cannot interrupt a procedure with an event, a state effectively defines the granularity of interruption within a modeled domain. So if you have an activity you might like to interrupt, you should break it up into a sufficient number of states. Conversely, a collection of actions that won't be interrupted can be grouped into the same state. (You can put the actions into an object method and just invoke it from one or more states to keep from drawing giant state rectangles!).

The modeler can't just assume that a state will be busy for a short or long time and just hope it completes before some other parallel activity. "This action should be really fast, so let's assume that it finishes before this other thing..." This lazy approach will not be rewarded kindly in a platform variant world! If a procedure must finish before or after some other activity, it needs to be explicitly coordinated with interlocking states or some other definitive mechanism.

If the modeler is not careful, a set of models might work fine during simulation and then break on a target implementation running at a different speed. Even worse, a system may work fine both in simulation and on the initial target platform. Then, some day, the platform changes either with new hardware or improved efficiency and *then* the models break! Platform independent testing puts the burden of careful synchronization and clear understanding of the synchronization rules on the modeler. That said, it is much easier to solve and test these problems in Executable UML than it is at the implementation level. And models that pass these tests should prove more robust and resilient than the brittle alternative frozen around a single platform.

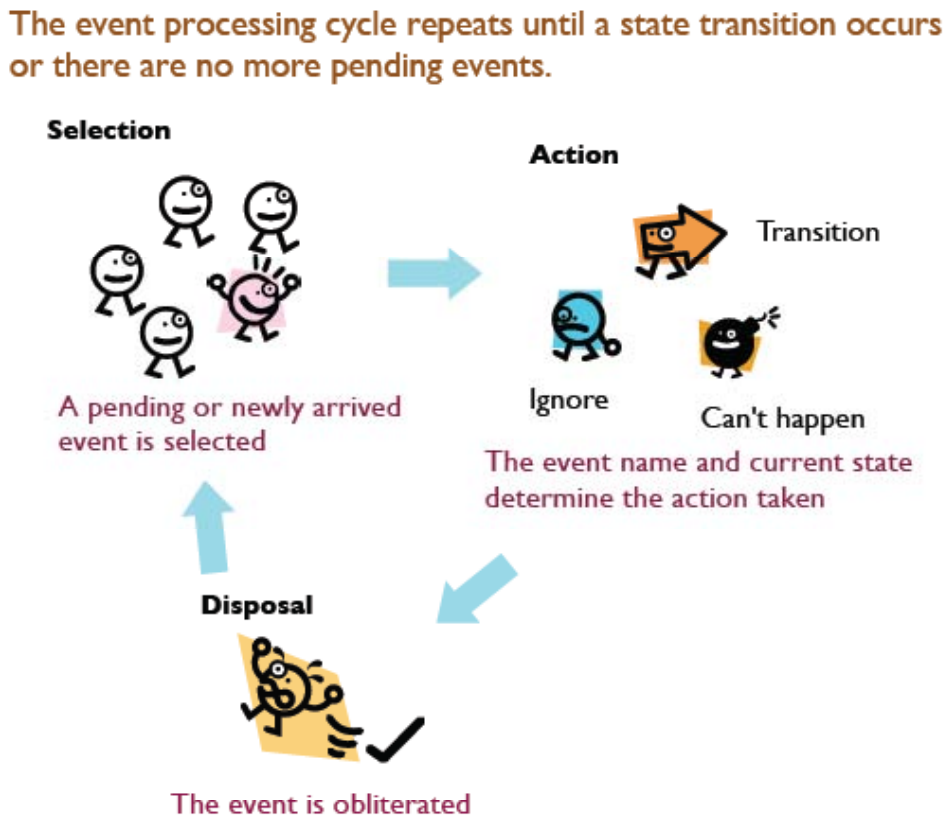
Implementation notes

While a procedure cannot be interrupted by an event at the model level, the MVM may suspend incomplete procedures to process system interrupts and then resume again. This is analogous to your desktop operating system swapping out your spreadsheet application momentarily and then swapping it back without you realizing it.

All Events are Processed and Discarded

If an object is waiting, events are processed as they occur. If one or more events are collected while an object is busy, they are processed one at a time once the object finishes executing its current procedure. In either case, each event is subject to the same event processing cycle.

The cycle consists of three phases: selection, action and disposal. For a waiting object, the event processing cycle is triggered when the next event occurs. This cycle is applied once for each event until a state transition occurs or there are no more pending events.



Selection

If there are pending events, the MVM (not the object) selects and presents one to the waiting object. If not, the MVM will present the next incoming event.

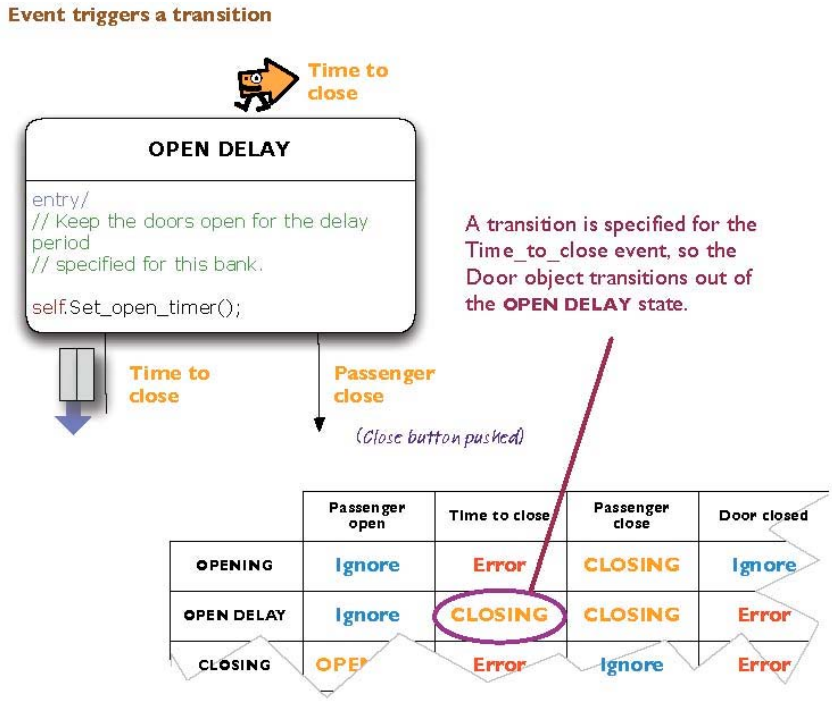
You may be surprised to learn that pending events are not necessarily queued. They can be, but it doesn't make any difference as far as the models are concerned. When event ordering is critical, it must be modeled explicitly without hoping that the MVM picks the right event. So the MVM is free to use whatever algorithm it likes to choose among pending or simultaneously occurring events. In practice, pending events are typically queued. This is a nice split between the MVM's and the modeler's responsibilities. The modeler shouldn't be worrying about how the MVM accomplishes its job as long as the synchronization rules work. Event prioritization, event ordering and granularity of interruption can always be managed with a properly synchronized and domain partitioned set of models without relying upon any special MVM magic. Novice modelers often demand special features from the MVM that would unnecessarily complicate the synchronization rules instead of just thinking the problem through and building the correct models. Event prioritization a commonly requested superfluous feature.

Action

The processing of an event in a given state is defined by the modeler as exactly one of the following choices: **transition**, **ignore** or **error**. Error" is also called "can't happen" or CH in the Executable UML community.

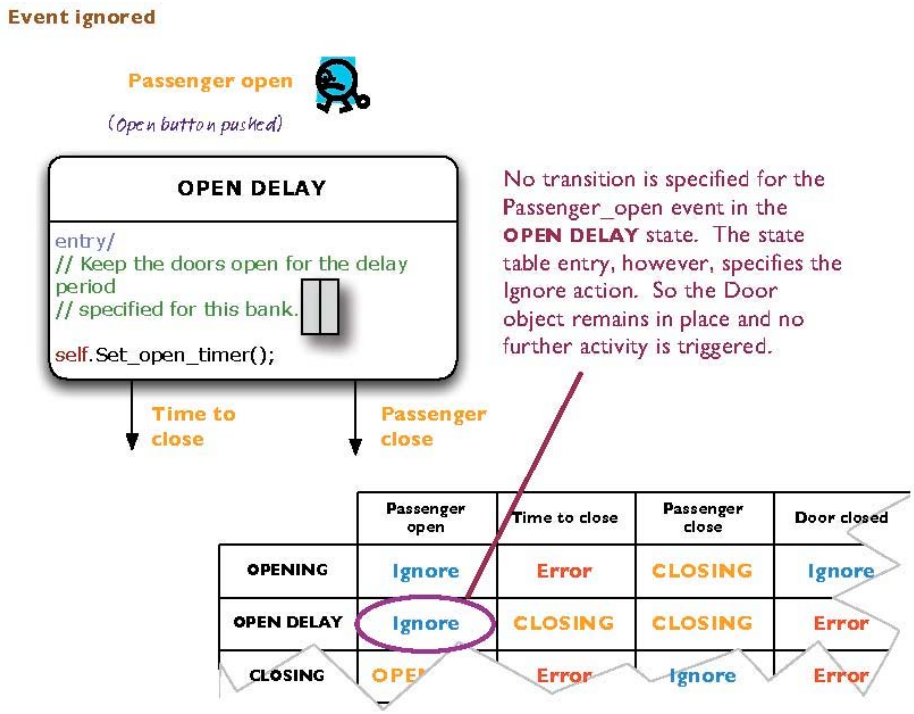
Transition

If a transition has been specified for the event, then the object will follow that transition to the destination state. Any parameter values supplied with the event will be made available to the procedure in the destination state.



Ignore

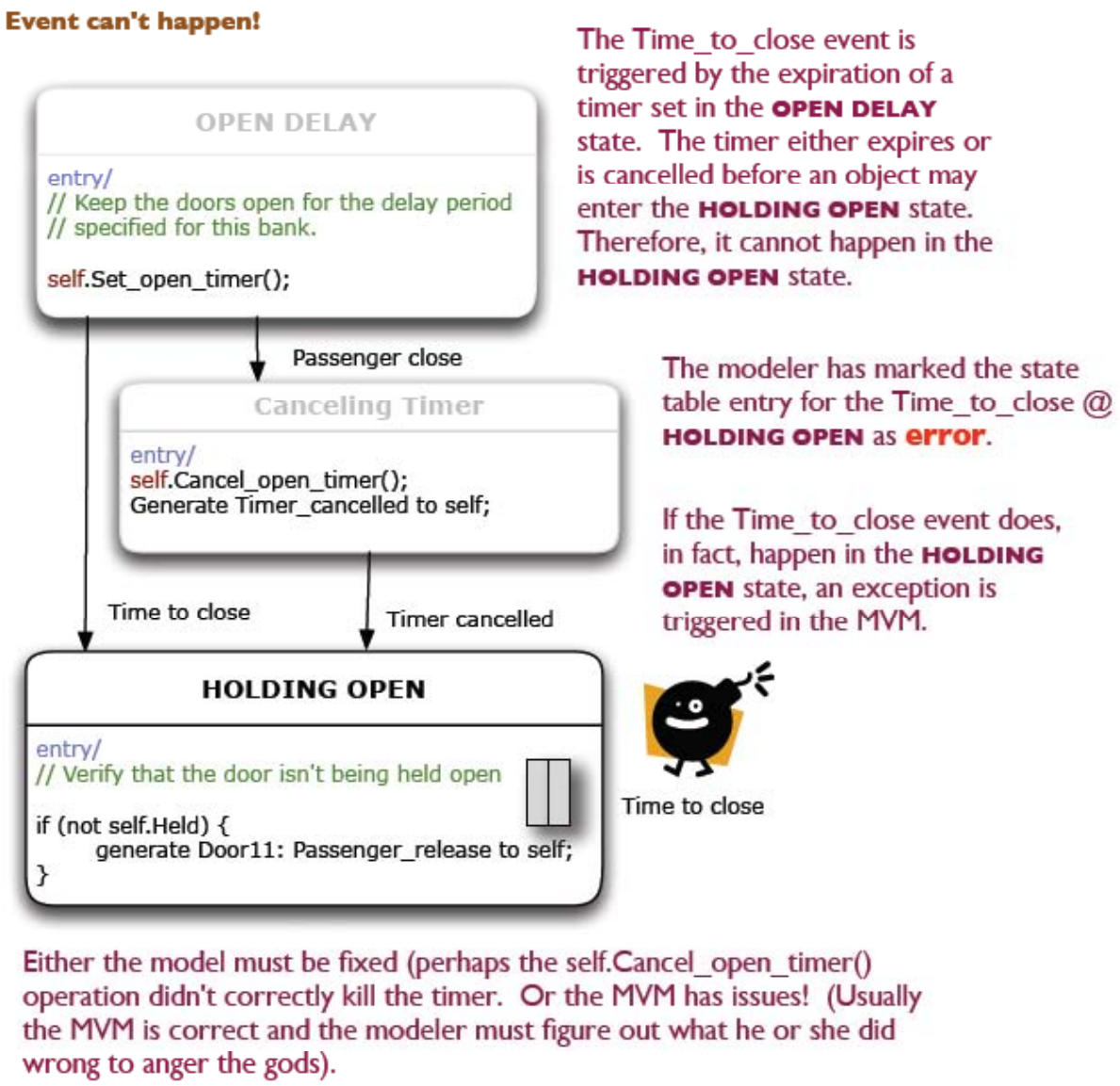
Otherwise, if **ignore** has been defined in the state table (not visible on the statechart), no action will be taken. The object remains in the current state. Imagine the response to someone pressing a button multiple times to call the elevator cabin to a floor. After the initial call is registered, but before the cabin arrives, it is safe to drop any subsequent presses.



Error

Finally, if **error** has been defined in the state table (again, not visible on the statechart), an error response is triggered in the MVM. This is similar to an exception in program code that bubbles up to the interpreter or run-time environment without being caught. A standard response to **error** will be designed into the MVM appropriate to the platform. The exception response of a Mars rover, an embedded cardiac pacemaker and an inventory control system will differ widely

I should probably point out that the state table is not part of UML. It is, however, necessary to fully define responses for each event-state combination. In practice, the process of carefully filling in each cell of the table will reveal several holes in an apparently complete statechart. Statecharts are great for identifying and understanding lifecycle behavior patterns. State tables are critical for ensuring completeness and directing attention to logical blind spots.



Disposal

The event is discarded. Once processed, an event is gone forever as far as the MVM is concerned. The modeler is, however, free to register the occurrence of an event or just the parameter data delivered, for later reference, by entering or updating data in the class model.

Consider, for example, the modeled event “Seismic Activity (Magnitude:2.7, Epicenter:San Francisco,Time:09:24)” which could trigger a state procedure that creates a new object in the Earthquake class and sets the corresponding attribute values. The general rule is that if you want a fact to persist, you need to put the appropriate data into the class model. Taking advantage of this fact can greatly simplify statecharts that attempt to field the same event in every single state for fear of missing it.

When a signal is delivered to an object busily executing a procedure, the MVM creates a corresponding event and holds it. If more events occur, they are also held. Eventually, the procedure completes and the MVM presents one of the held events to the object. The object will process this event as t transition-ignore-error. In all three cases, the presented event will be destroyed in the process.

Signals and Events

When a signal is delivered to an object, it triggers an event of the same name. In proper UML speak, a signal is sent and an event occurs. Since the only events that occur in Executable UML are triggered by signals, we often use the terms signal and event interchangeably.

If the event triggers a transition, again the event disappears with any supplied parameter values passed to the procedure of the subsequent state. If more than one event was pending, it will continue to be held until completion of the next procedure. Does this mean that held events may be processed several states downstream? Yes! Events persist until they are processed, at which point they are destroyed. (This suggests yet another technique to avoid having to field the same event in every state of a state-chart). Both techniques will be employed in the upcoming door statechart example, so keep scrolling!

Events are not necessarily prioritized or queued by the MVM. In other words, the modeler should not count an any such thing. Events happen. If you need events to happen in a particular order, or if certain events must supersede or preclude lesser events, you need to manage this in the synchronization of your states. Remember that events are held, not necessarily queued.

The MVM sees two kinds of events. Self directed and non-self directed. When an object sends a signal to itself, usually as part of an if-then action or just to trigger an advance at the end of a procedure, it results in the occurrence of a self-directed event. Collaboration among other objects of the same or different class happens when a signal is addressed to any other object.

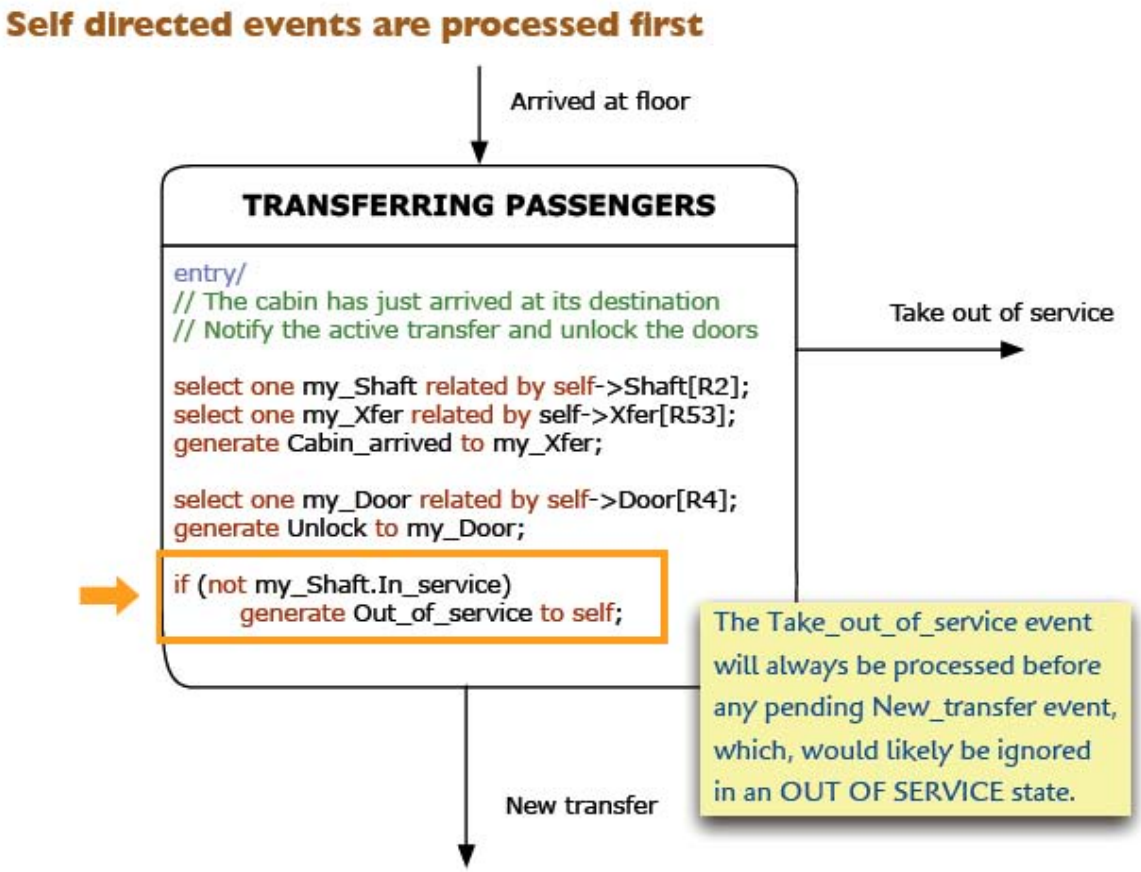
Self directed events are always presented to an object before any non-self directed events are seen. For the moment, let’s consider only non-self directed events.

The MVM saves these as they occur. There is no queuing requirement specified in Executable UML (nor is there a need for one). It is the modeler’s responsibility to make sequence explicit, not hide it in the MVM behavior. That said, most implemented MVMs queue incoming events as a matter of implementation convenience. Just don’t count on it.

Self-Directed Events Take Precedence

Now having said all this, there is one exception to event prioritization which applies only to self-directed signals.

You may have noted in the previous example that an object can force itself to advance (or re-enter the same state) by sending a signal to itself. This feature is especially useful when local decisions are made as was shown below:



Either signal generated as a result of the if-then-else test is relevant only if processed in the current state. In fact, you may notice that an object will never wait in a state like this because a self-directed, transition triggering event is guaranteed to be waiting upon procedure completion. (My own personal convention is to use initial caps to title such states as opposed to all uppercase for potential dwell states. It would be nice if the supporting tools would use color or some other visual feature to make this distinction). So, the rule is that an object never sees a non-self directed event before all pending self-directed events have been processed.

A self directed signal is sent from an object to *itself*. This is not to be confused with an event sent from one object to another object of the same class!

Note to methodology/MVM definers: I think this is a bit of turbulence on an otherwise aerodynamic system. Why not just replace self-directed events with the concept of forced transitions? Then we could dispense with event priorities entirely. Am I right?

Signal Delivery Time is Unknown

When a signal is generated, it is sent immediately. Signals are never lost and delivery (in the form of a corresponding event) is guaranteed. The delivery time, however is unknown since this is platform dependent. So the modeler should not make assumptions that rely on a signal arriving within an absolute time window. Having worked on a number of video frame rate systems, I can hear the collective gasp out there! In such systems the underlying MVM on the target platform must provide mechanisms to ensure that hard time boundaries are respected and the corresponding events are triggered on time. In other devices, critical interrupts must be

handled immediately. Some procedures may have to finish within critical time windows. All of this is transparent to the model level, however. This is why we don't have a one-size-fits-all MVM! Each class of platform requires its own MVM.

Implementation note

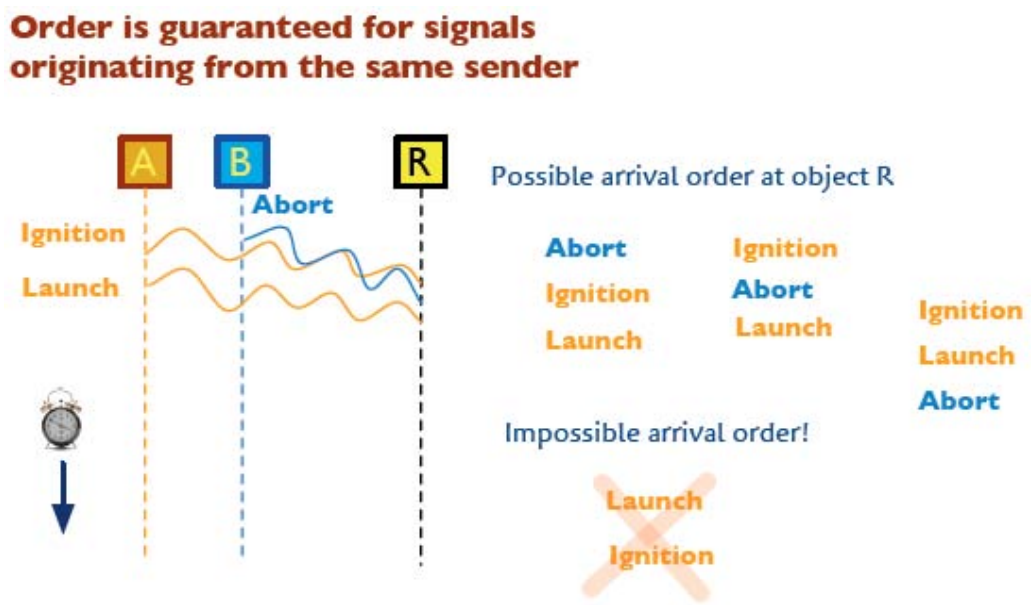
Now you may rightly assert that signals do get lost in the real world. Yes they do. But as far as the modeler is concerned they don't. The MVM is responsible for ensuring delivery. So the modeler should not add model logic to compensate for lost signals. This is analogous to assuming that a function call in the C programming language will, in fact, invoke the function. You don't write extra C code to recover in case the function calls don't get made. Of course, you might not like the return value, but you assume that the function actually does get called.

Of course, the MVM might screw up and lose a signal. For that matter memory might get corrupted, again, not of concern to your models. It is the MVM's responsibility to recover, not your application models. Typically this will involve rebooting your domain or resetting it to an earlier image. The range of soft, cold and hard restarting supported by an MVM is something that varies from one platform to another.

Signals from the Same Sender Arrive in the Order Sent

Without a global clock, there is no way to resolve the time at which two signals from different senders (objects or external entities) are issued. Consider two senders A and B. A sends **ignition** and B sends **launch** at the same time to a recipient object R. But A and B each have their own local, unsynchronized clocks. So even if each signal is time-stamped by its sender, the "same time" cannot be resolved without a global clock. So R will see either **ignition** -> **launch** or **launch** -> **ignition**.

Signals from the *same sender* will, however, be seen in the order sent. This is because timestamps from a single sender's local clock can be resolved by the MVM. If sender A issues **ignition**, **launch** and B sends **abort**, object R may see any of the following: **abort** -> **ignition** -> **launch** or **ignition** -> **abort** -> **launch**, or **ignition** -> **launch** -> **abort**. In each case, **ignition** is seen before **launch** by object R.

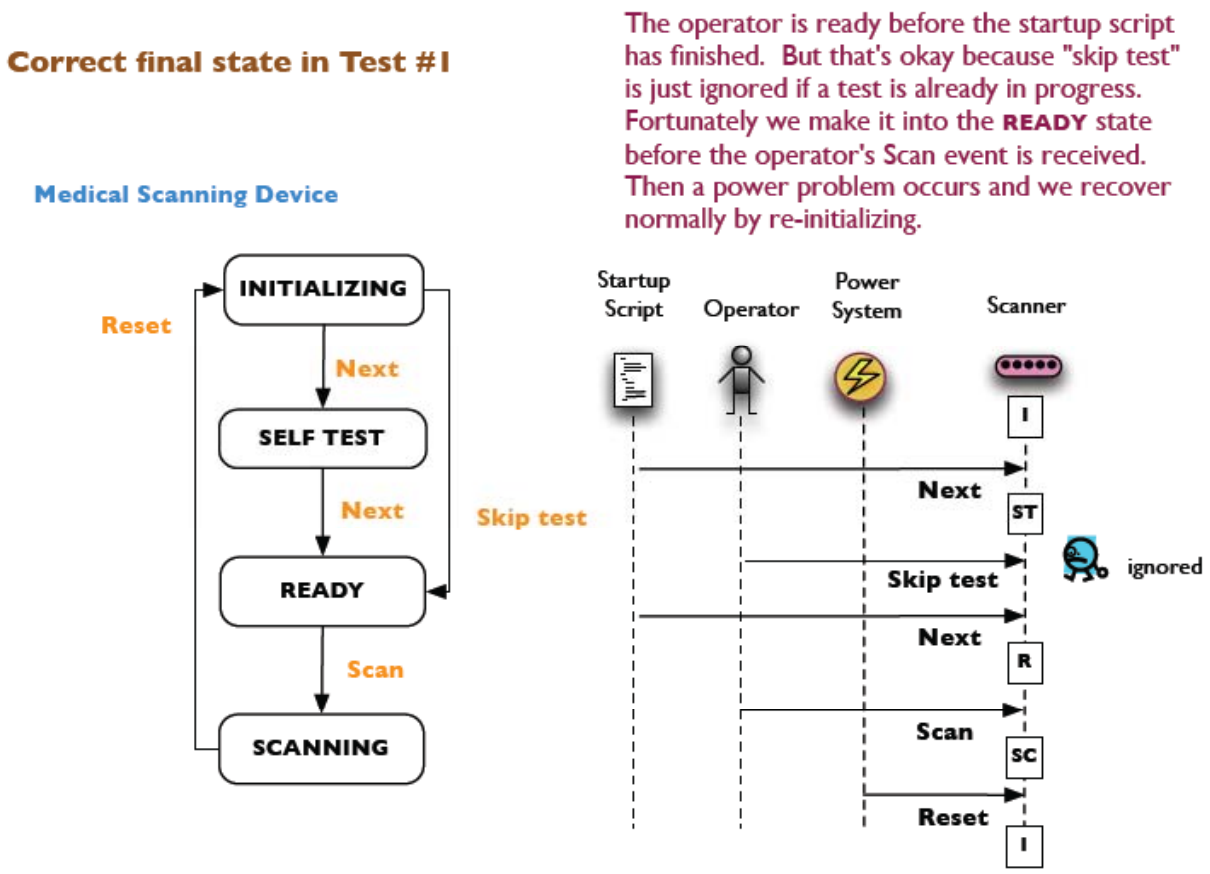


This is not unlike the situation where two friends e-mail you simultaneously. The internet does not run on global time and will deliver the messages in either order depending on various latencies. On the other hand, an individual message that you receive contains perfectly ordered text. The packets from a single sender are kept in order even if they arrive sporadically thanks to local timestamps.

The power of this highly distributed time model is that you can build models that reliably interface with an asynchronous and often chaotic world. And once tested against the Executable UML rules, these models will work on any platform. But with great power comes great responsibility. The modeler must now test against chronologically variant scenarios. Here are a few examples to give you a sense of what is involved.

Same events, but one test passes while the other fails

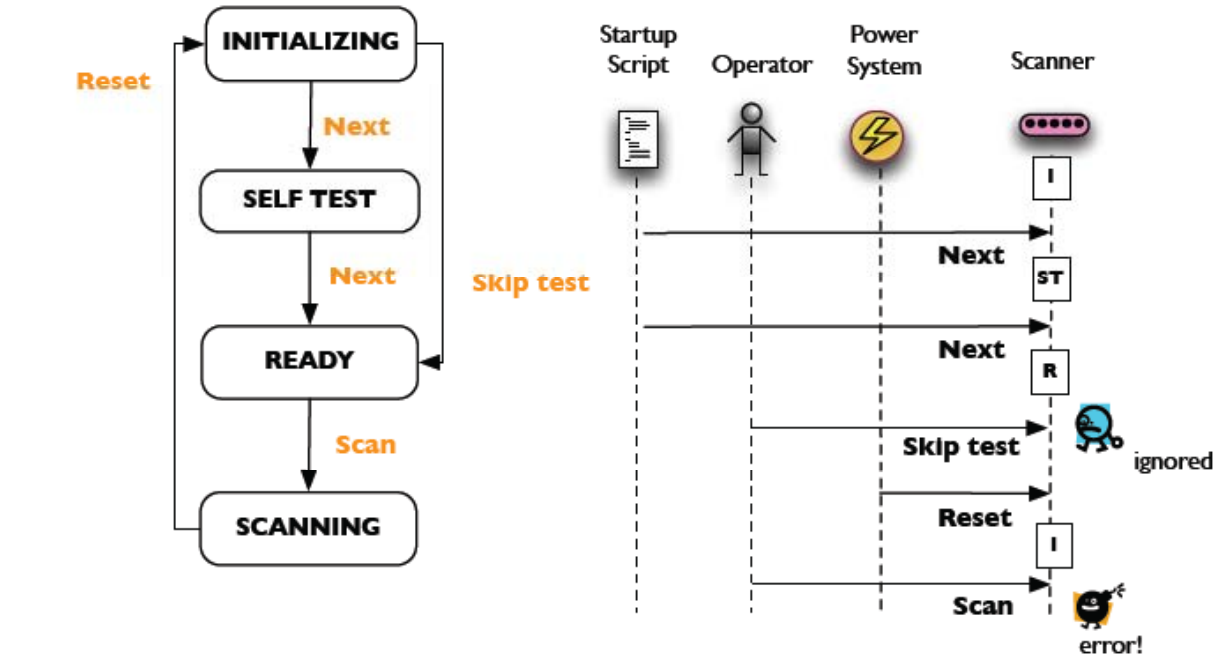
In this first example we have a medical scanning device that interfaces with a startup script, a human operator and a power management system. The operator will want to skip the self test phase and just start a scan. But the startup script is running and at any point the power system may need to reset the machine.



Assuming the device starts out in the INITIALIZING state, we will see the following state sequence: INITIALIZING, SELF TEST, READY, SCANNING and INITIALIZING again. So the idea is that the startup script is still running when the operator's commands arrive. Then, during the scan, the power system re-initializes the device. Lack of synchronization between the operator, startup script and power system is troubling. But no error occurred in this Test #1. Let's try again, possibly on a different platform (or with different tuning in the simulation) and see what happens.

Same events, but test 2 fails!

Again, the startup script is too fast for the operator (even faster than in test #1, and the Skip Test event is still ignored. But the power reset happens early as well and the Scan command causes an irrecoverable error! Same events, but different sequence due to different platform latencies.



The exact same signals are sent as those in Test #1. This time, though, the startup script executes a bit faster so the 2nd Next event is seen before the operator's Skip Test event. That's okay, because the READY state is configured to ignore the Skip Test event if it occurs. But the Reset event occurs before the Scan command this time. The INITIALIZING state is not expecting a Scan event and registers an error when this happens. So the second test gives us an error!

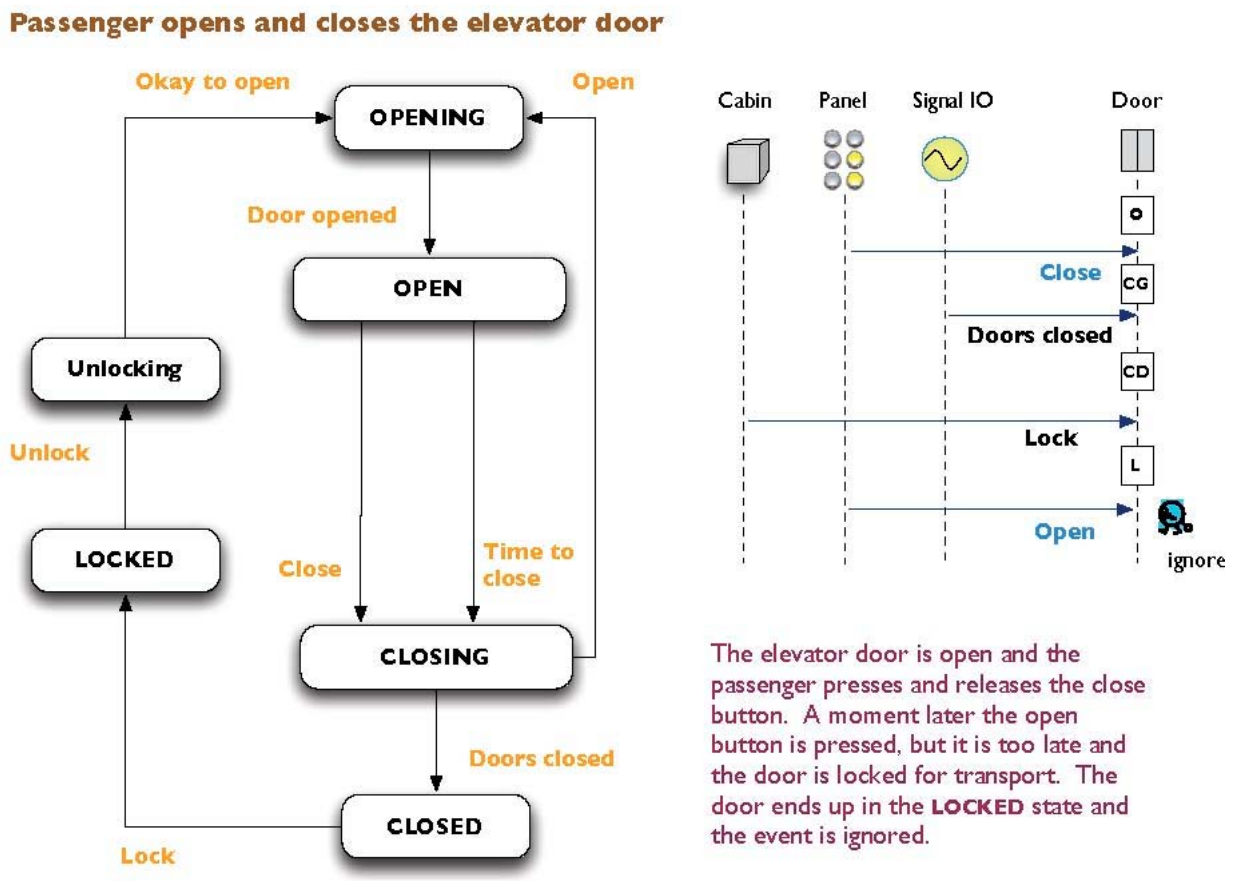
There are many ways to resolve the discrepancies between tests 1 and 2. We could use a combination of events and interlocking states to keep the three external entities from stepping on one another. For example, we might queue operator commands (as objects) until the startup script is finished and then generate events as we dequeue the operator commands. We could also have the operator interface hold in some state until triggered from the READY or newly added WAITING TO INITIALIZE states. Or we could just add lots of **ignore** responses to the state table to make the state model more resilient. Be careful though, the practice of just ignoring anything you don't care about willy nilly leads to oblivious state models that don't synchronize well with each other and the outside world. Use the **ignore** response sparingly! My approach is to initialize all state-event responses to error, when I am building a new state model, and then to selectively and strategically insert **ignores** as required. Each **ignore** will carry a strong justification, which should be included in the documentation. (If you use a spreadsheet like I do, sometimes, you can add comments to the table cells).

The main point here is that the modeler must remember that signals from different senders are synchronized only if explicitly modeled. Only these signals from the same sender are guaranteed to arrive in order.

Strategic use of the ignore response

In this next example it is unrealistic to coerce the input from multiple senders into a convenient sequence. We instead want our state chart to respond resiliently while accurately synchronizing with another statechart. This is our elevator door-cabin problem mentioned at the beginning of this article. We want to ensure that the cabin never moves unless the door is closed. But we can't prevent a passenger from pushing the open button just at the point that the cabin starts to move. Furthermore, if the cabin wants to move at the exact time that the user pushes the open button we can't know which event will be seen by the door first. We can solve this problem by adding a LOCKED state to the door statechart and employing the **ignore** response in all but the CLOSED state.

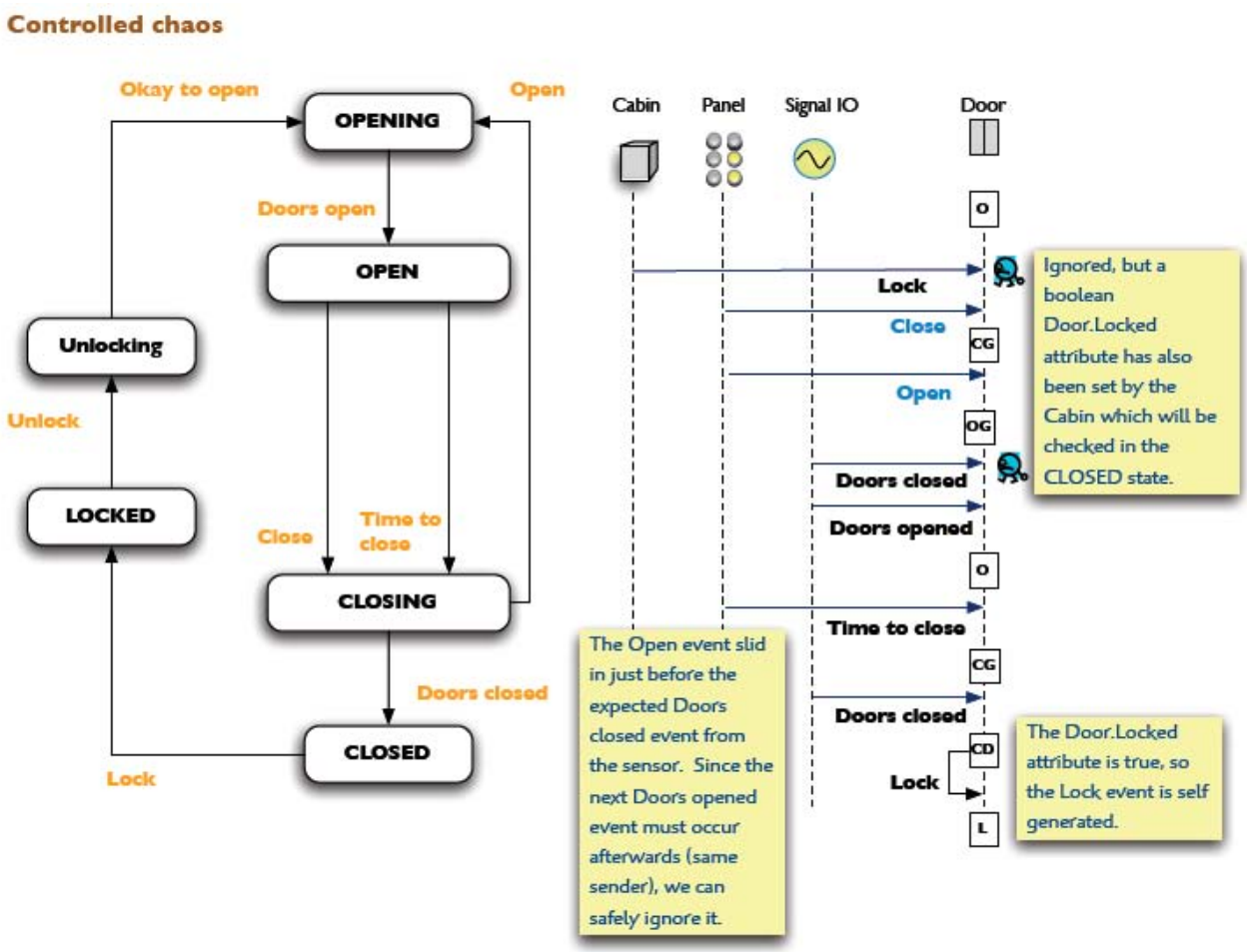
An elevator door will receive unsolicited and uncoordinated open and close requests from the passenger and opened and closed events from the door sensors. There will, however, be coordination to ensure that the cabin does not move until the doors are closed.



The door starts off in the OPEN state when the passenger presses the close button. The corresponding event pushes us into the CLOSING state where we wait until the sensor reports back with a Doors_closed signal. Now in the CLOSED state a Lock signal is received from the cabin (it wants to move). The door enters the LOCKED state where user open/close requests will be ignored. In fact, the user tries to open the door, but too late, the event is ignored.

Naturally, this is only one of many possible event arrival sequences.

But what would have happened if the timing was slightly different?



Things appear to go wrong from the outset. The cabin is ready to move but may not proceed until the door is in the LOCKED state. The Lock event is received, but ignored by the door object since it is currently in the OPEN state. According to our rules, the Lock event is then discarded. Fortunately, though, the cabin (trust me, it did Or don't and just download the Elevator Case Study [5] and see for yourself!) set the Door.Locked attribute to true in the same procedure that sent the Lock signal. We will refer to that attribute value later.

The operator pushes the Close button and we see the corresponding event and transition to the CLOSING state. At this point we are expecting the Doors_closed event from the door sensor. But at the exact moment that the doors fully close, or perhaps a split second before, the passenger hits the Open button. The passenger and door sensors do not coordinate with one another, so the Open event could snake in first and put us in the OPENING state. The doors are commanded to open in the state procedure and, upon completion of the procedure, the door object sees the tardy Doors_closed event. Note that there is no possibility of the impending Doors_opened event getting here first as it is from the same sender and must arrive *after* the Doors_closed event. And we can't exit the OPENING state until we have the Doors_opened event, so, if the Doors_closed event did not get processed in the CLOSING state, it will get handled in the OPENING state. And the point is moot in the OPENING state, so the Doors_closed event is safely ignored. (Assuming we set this correctly in the state table which I did not do the first time!)

We eventually transition into the CLOSED state and execute its procedure (not shown) which checks the status of the Door.Locked attribute. Since it is true, the door object sends a Lock signal to itself (again, not shown). This pushes the door object into the LOCKED state. The attribute will be reset to false later in the Unlocking state. The cabin and door statecharts are

coordinated so that there is no possibility of a write conflict on the Door.Locked attribute. Download the models [5] if you are interested in seeing more detail.

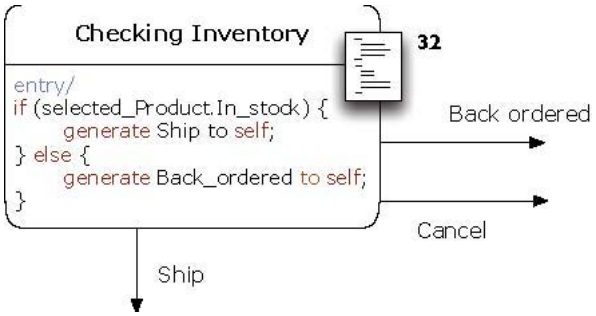
It should be clear by now that a firm understanding of the Executable UML synchronization rules is essential to addressing real world concurrency. Notation, by itself, doesn't get the job done. It should also be clear that executable modeling keeps the modeler honest. You are no longer allowed to leave the hard problems for the programmer/designer!

Resolving Ambiguity

Now let's go back to the original examples and interpret them using the Executable UML synchronization rules.

Example 1: Busy when event occurs

Order 32 is executing the Checking Inventory procedure when a Cancel event occurs. What happens?

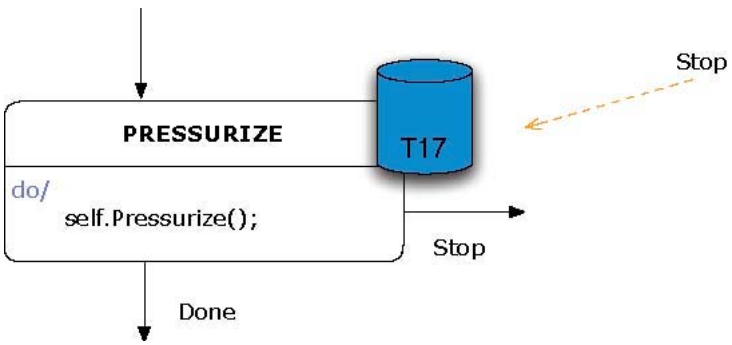


We know that a busy object does not see incoming events. So the state procedure will execute and produce either the Ship or Back_ordered event. Assuming the Cancel event occurs during this very short time window, it will be held by the MVM. We also know that self directed events are selected and presented by the MVM first. So Order 32 will always exit on a Ship or Back_ordered event, since each triggers a transition. The Cancel event would be retained in the event pool until some future state where it would most likely be ignored.

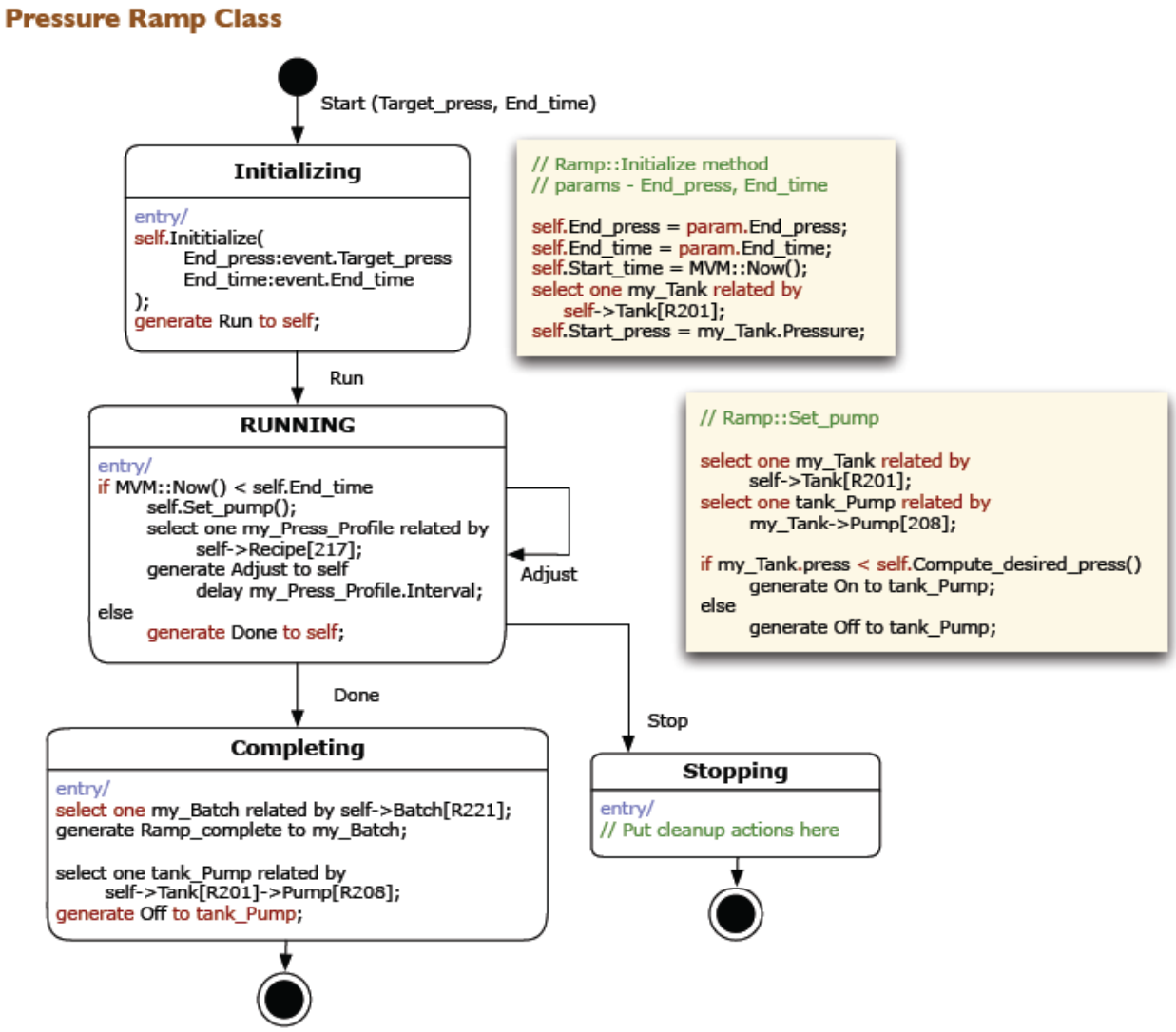
Since we know that we can't interrupt a procedure with an event, we wouldn't build the statechart this way. We would probably intercept the Cancel event before or after the Checking Inventory state. Another idea would be for some operation or external function to set a boolean attribute on the Order class such as Order.Cancelled. Then, in a subsequent state, we could check it: if self.Cancelled, generate Cancel to self; In any case, we would certainly remove the Cancel transition from this example.

Example 2: Granularity of interruption

To answer our original question about granularity of interruption for a continuous process, we can break this single state example into discrete components.



This will result in a more controllable but bigger statechart. Don't worry though, because we can then package the more complex statechart in a service domain and just start and stop it from our application domain. This is one of many easy ways to avoid the unnecessary complexity and arbitrary leveling of statechart hierarchies. First, we will define the behavior of a Pressure Ramp class.



The Pressure Ramp is created by specifying a target pressure and some future time. It immediately starts up and adjusts the pressure by turning a pump on and off until the completion (end) time.

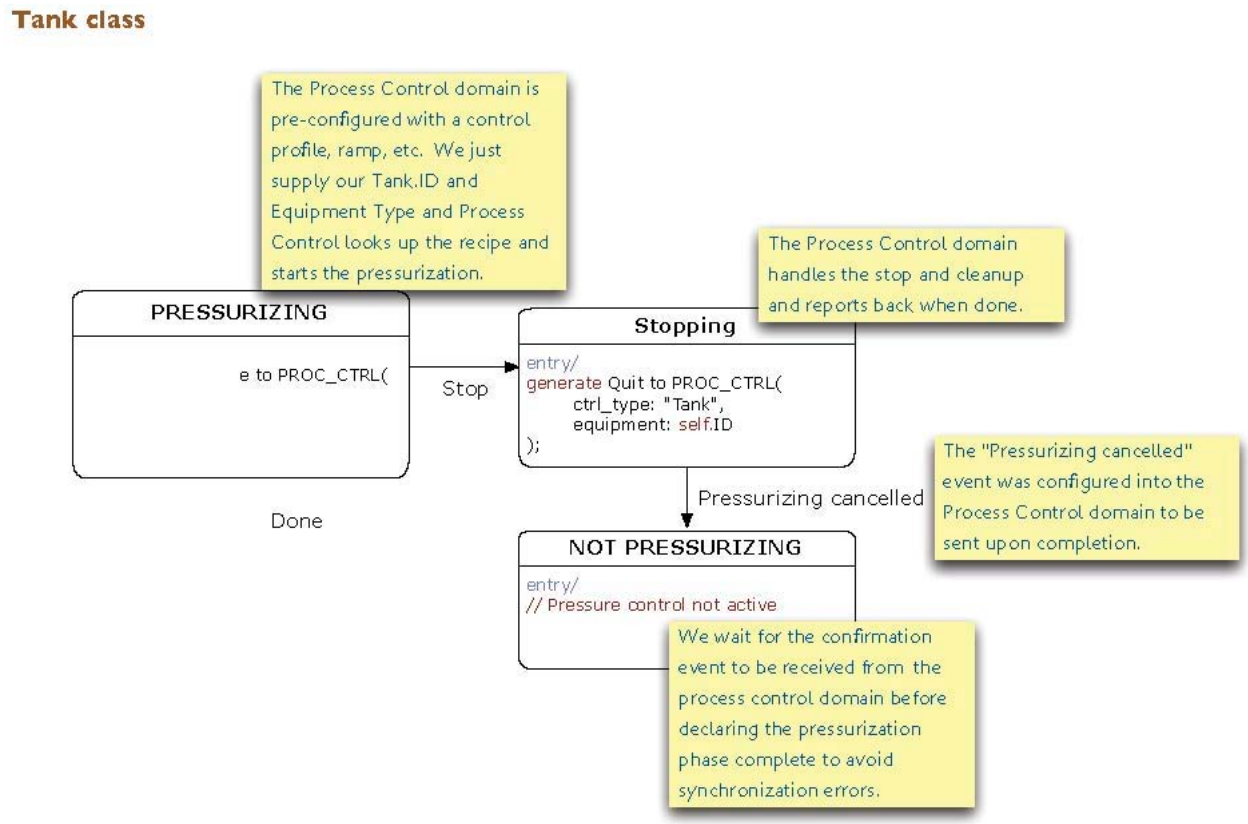
The procedure's actions are organized into methods, shown on the right, which are invoked from the appropriate state. This keeps our statecharts more readable, printable and testable. State

actions should focus on control with the data processing and computation encapsulated in the methods.

As you can see, we can interrupt the pressure ramp with the Stop event. If we are busy executing the RUNNING procedure, this event will be pooled until after we generate the delayed Adjust event. At this point we see the pending Stop event and transition out cleanly. This is a nice solution since we don't corrupt any data on the way out. Keep in mind that the MVM can shut us down even in the middle of a state procedure if, say, someone hits an E-Stop button. Normally, we would gracefully interrupt by sending a Stop signal, though. So it is the modeler's responsibility to anticipate real world interrupts and provide the appropriate cleanup mechanism if the interrupt can reasonably be handled in the application domain.

The Pressure Ramp statechart may be more eloquent, but the original example was concise and intuitive. We should, and can, get the best of both worlds. First we recognize that the Pressure Ramp concept is not specific to pressure. If we wanted to control temperature instead, we could globally replace "Press" with "Temp" on our statechart. So the renamed Ramp class could be subsumed as part of a generic process control service domain that could then be configured and directed from our higher level (client) Pressure application domain.

Along with the Ramp, we would have loaded appropriate control profile data into the Process Control domain, and bound it to corresponding entities in the Pressure Application domain. Now we can create the following simplified statechart in our application.



The entry procedure in the tank's PRESSURIZING state triggers pressure control feedback activity in the Process Control service domain. The triggering event does not specify a ramp since the whole purpose of a domain separation is to insulate the analysis and implementation of one domain from another. So we just specify our type "Tank" and ID value knowing that the bridge to the Process Control domain maps to the relevant content, a ramp and any control profile data in this case.

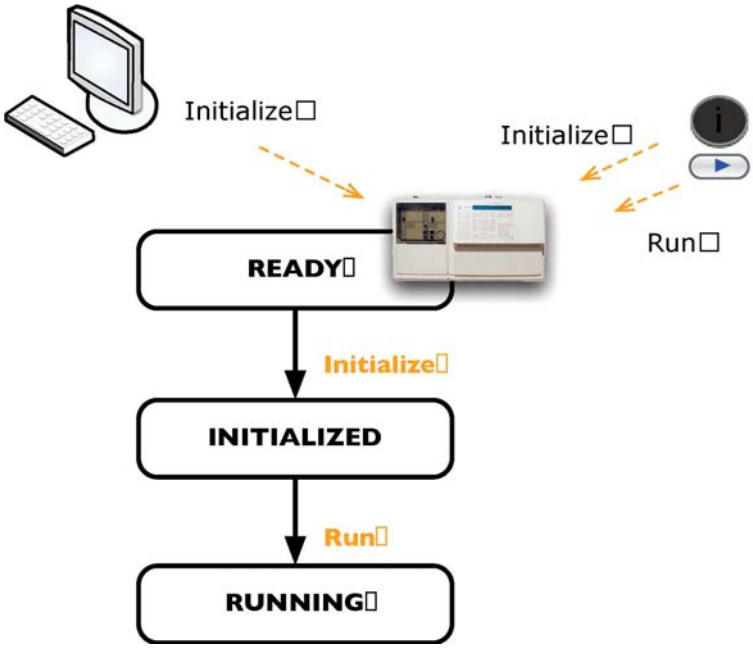
We then sit in the PRESSURIZING state until we get a Done event (triggered by completion of the Ramp or whatever mechanism is running in the Process Control domain). Alternatively, a Stop event occurs which puts us in the Stopping state where we tell the control mechanism to quit. (Another way to do it is to route the Stop event directly to the Process Control domain and just have a Stopped event bubble up to the Application domain. Either way, we end up in the, for lack of a better name, NOT PRESSURIZING state.

Now the application reflects the relatively high level view of the Tank, the Process Control domain handles ramps, profiles, control curves and such with an even lower level Signal IO domain managing the analog/digital interfaces to the sensors and actuators. Bridge'em all together and we have ourselves a system!

So with the availability of domain packaging and clear synchronization rules, we dispense with the need for special **do/** semantics in Executable UML. The **entry/** procedure is all we need. You can define an Executable UML with exit/ procedures only and might be just as good. There are many equivalent state machine formulations (Mealy/Moore), etc out there. The point is to keep it simple. Inclusive use of both entry/ and exit/ actions and incorporation of state history, while we're at it, is a recipe for needlessly confusing models

Example 3: Sequencing events

In this last example we have a scientific measurement instrument commanded from both an LCD button panel on the front of the station and via a remote workstation connected on a network. A human operator presses the Initialize and Run buttons in sequence. Simultaneously, another Initialize command arrives from the network. What happens?



We know that the Initialize event from the front panel will be seen before the Run event. But the Initialize event from the Network could slip in at any point. An object sitting in the READY state will certainly see one of the two Initialize events first and then transition to the INITIALIZED state. At this point, either the Run event or the second Initialize event will be seen. If **ignore** is specified as the response in the INITIALIZED state (likely), the second Initialize event will be dropped, then Run will be seen followed by a transition into the RUNNING state. If, instead, the Run event is detected second, the object will transition to the RUNNING state where the 2nd Initialize must be processed somehow (**transition**, **ignore**, **error**).

Now is this the desired behavior? Without the requirements at hand, you don't know. But you do know precisely how this model will behave given a specific scenario.

Making your own Rules

The Executable UML described in this article is documented in a number of books and supported by multiple vendors. More importantly there are at least a dozen MVMs and model compilers that support the rules described in this article (including two running on my laptop). See the tool references at the end of this article for more information. Additionally, several more specialized MVM's and model compilers have been built which are custom built internal to companies that deliver embedded and distributed systems.

While Executable UML defines a robust, simple and time tested set of rules, you are free to invent your own. Here are some pro's and con's to consider.

Pros

+ Adapt to special circumstances in your application or platform. However, you'll probably get more leverage by directing your talents at a better model compiler / MVM implementation.
+ Adapt to the capabilities of non-Executable UML modeling tools.

Cons

-There is a huge investment of time and effort to ensure that the rules are internally consistent and have
no unforeseen consequences. Plus you'll have to write yourself a book describing the rules for any future modelers.
-If you change the timing and synchronization rules, you lose compatibility with any existing Executable UML model compilers, model level debuggers and MVMs. So you will have to build these as well.
-All my example models will break if you change the rules! So you'll need to write your own tutorials for prospective modelers ;)

If you do decide to define your own Executable UML, the OMG has recently published a guide to profiling UML accordingly [8]. Good luck!

The important thing is that you somehow arrive at a set of timing and synchronization, data access and other rules agreed upon by both the modelers and the platform developers. Ultimately you need to decide whether you want to devote your ingenuity to modeling the application, defining a new methodology or designing an efficient platform. I've made my choice to focus on the first item since there seems to be plenty of bright folks focusing on the other two.

Summary

Executable UML adds executability to UML by introducing a number of features including platform independent time and synchronization rules. Other features necessary to make UML a full fledged development language are a precise underlying data model for the class diagrams and a rigorous action language.

A key characteristic of the time and synchronization rules is that concurrency is the default paradigm. We assume that each object operates in its own local timeframe. Procedure

sequencing and state synchronization must be explicitly modeled and tested. You could rightly say that Executable UML comes with no “artificial sequencers”. This makes it possible for a single modeled domain to run on a wide variety of target platforms. There is no need to tweak the models to take advantage of added available parallelism since the models are already tested to run with maximal concurrency. The more parallel your platform, the more efficiently the models may run.

An MVM is necessary to run Executable UML models. Ordinarily, the requirement for a VM implies a significant burden on the implementation. In reality, though, the MVM can be custom built for various *classes* of platform. (One common misconception is that a separate MVM must be created for each platform. Not true, especially with a data driven MVM design!) The MVM implementation on each platform class, however, will be quite different. A highly embedded MVM may optimize for small event queue sizes, assume fixed instance populations and even provide a way to explicitly bind instances to absolute memory locations. A highly distributed MVM will may manage a sophisticated threading, fault tolerance and multiprocessor scheme.

But the same time and synchronization rules are supported regardless of MVM implementation. Some MVM’s may support a variety of marking (The “marking model” is an MDA concept. See the MDA Distilled book [2] for a good explanation.) features so that various model components in an Executable UML domain can be compiled more efficiently. Some model compilers take into account initial data populations in addition to the models and markings to fully optimize the implementation.

The MVM of a desktop simulation environment, in particular, will be quite different than that of a highly distributed or highly embedded platform.

Since synchronization problems can be definitively resolved, serious models can be built. This also requires a rigorous action language to define the state procedures. It’s a topic for another article to be sure. But parallelism available at the state level can be (and is) carried into the procedures so that multiple actions run in parallel as data dependencies permit.

With the support of Executable UML, intellectual effort can be focused on resolving complex systems at the application level. The resulting distillation of intellectual property is highly portable. Organizations that produce spin-off products from an initial design should benefit greatly.

On the down side, a whole new set of development skills and mindset is required to build Executable UML models. The ability to focus on an application without getting caught up in issues that are the domain of the MVM and platform is difficult for many developers. Also the ability to think and test in concurrent terms takes a bit of practice. That said, the days of sequential thinking in system design are long gone.

Published Resources

[1] How to Build Articulate UML Class Models, OMG uml.org and Google Knol, Leon Starr, 2008

[2] MDA Distilled, Principles of Model Driven Architecture, Mellor, Scott, Uhl, Weise, Addison-Wesley, 2004, ISBN 0201788918

[3] Executable UML,A Foundation for Model Driven Architecture, Mellor-Balcer, Addison-Wesley, 2002, ISBN 0201748045

[4] Time, Clocks and the Ordering of Events in a Distributed System, Leslie Lamport, Communications of the ACM 21, 1978 (yes, really, 1978!)

[5] Elevator Application Executable UML Case Study, Leon Starr, Model Integration LLC, 2001, 2005

[6] Model Driven Architecture with Executable UML, Raistrick, et al, Cambridge University Press, ISBN 0521537711

[7] Executable UML, Wikipedia entry

[8] Semantics of a Foundational Subset for Executable UML Models, 2nd Revised Submission, OMG, Aug 2008 (I have not been able to find this on the web yet, sorry). I had my copy e-mailed to me by Steve Mellor. Will update the online version of this article as soon as I find it.

[9] Semantics of a Foundational Subset for Executable UML Models (FUML), OMG, 2008. (Seems to be a subset of document [8]) http://www.omg.org/spec/FUML/

Tool and Vendor Resources

If you are interested in tools that support Executable UML, here are a few:

Nucleus BridgePoint, Mentor Graphics, http://www.mentor.com/products/sm/uml_suite/

iUML, Kennedy-Carter, http://www.kc.com/

Pathfinder Solutions, http://www.pathfindermda.com/

OOA Tool, Kavanagh Consultancy, http://ooatool.com/Products.html

For my own work, I use BridgePoint mostly. But I work with whatever tools my clients are using including Artisan, Rose-Rhapsody (IBM).

Advertisers in this issue

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php