Bringing stakeholders together through modeling

Evan Masters

emasters@critical-logic.com

Abstract

Written specs are often confusing, ambiguous, incomplete, or simply gargantuan and overly complex. This leads directly to defects being built into the business systems that they attempt to describe. In this talk, I will describe a method that enables enterprises to deliver high quality business systems in a repeatable, predictable fashion using technology, standards, modeling principles, and automation to validate and verify system behavior.

We have all heard that a picture is worth a thousand words, so why do we use so many words to describe things instead of using pictures? In this talk, I will describe how the act of creating pictures in the form of models to visualize the intended behavior of a business system brings together stakeholders of every part of the Software Development Lifecycle (SDLC). These models augment the documentation created to convey business needs to the development team. Models can eliminate ambiguity, clarify confusion, and fill in the gaps from incomplete specs. They can also provide a simplified view into the complex, making what may seem massive more manageable.

I have personally been involved in more than two dozen development projects where models were created as part of the design phase, and my organization has been involved with over 100 such projects. The models were designed with the intent of using them for Model-Based Testing (MBT), but they served a useful purpose to nearly all stakeholders involved in the business system development process. In some projects, the organizations were able to do more with their existing staff. In others, software quality metrics were increased. Still others were able to begin implementing automated testing using these models as guides.

In all cases, visualizing the information in the form of models creates a common understanding of the desired behavior of the business system under development, reduces the defects introduced into the development process, and brings stakeholders together to take charge of quality at every stage of the design and development processes.

Biography

Evan Masters is passionate about seeing customers succeed. With over 10 years' experience in the software quality assurance industry, Evan has helped companies around the globe overcome challenges and implement innovative new technologies in support of business goals and objectives. Working with companies of all sizes, from startups to Fortune 5 companies, and thriving on his ability to listen to and understand customers' real needs, he has equipped and supported teams with the tools necessary to take their capabilities to the next level.

1 Introduction

To meet the ever-increasing demands of delivering high quality software products and solutions at a pace matching rapidly evolving business needs, Software Quality Assurance (SQA) professionals need to leverage every resource at their disposal. An effective strategy SQA professionals can employ to ensure the quality of the product or solution is to engage with stakeholders at every stage of the SDLC, not just the Testing stage. A tool that can be used to implement this strategy is a diagram of their understanding of the intended behavior of the business system. This helps to prevent any misalignment in the understanding of the intended behavior across the various stakeholder groups.

Creating (good) models and diagrams to augment text-based definitions of the desired system offers many benefits which I will discuss in depth throughout this work. Creating models and diagrams allows SQA professionals to become engaged much sooner in the SDLC than typically realized, putting quality front and center from the start of the process. These models and diagrams give the SQA professionals a layer of abstraction that can be used to convey an understanding of the system being developed in a way that stakeholders of any technical or business level of expertise can understand.

As organizations grow and SQA teams improve their capabilities, models and diagrams can evolve with them. SQA groups can leverage the models created early in the SDLC to design and generate their tests. This process is known as Model Based Testing, or MBT. SQA groups who automate their test execution can leverage these models even further, exposing their automation frameworks at the model level. This process gives stakeholders at every level the ability to create automation scripts, regardless of their technical expertise. I will describe this evolution from diagram and model creation, to Model Based Testing, to automatic script generation below.

2 Benefits of Creating (Good) Models and Diagrams

Before I dive into my description of how creating models and diagrams can bring stakeholders together to improve software quality, I will begin with a question: What is the end-goal of software development? The most limited and least prescriptive answer is “to deliver quality business systems.” This begs the follow up question: “does software development always achieve the goal it sets out to?” The obvious answer is no, it does not. Why is this? In practice, there are numerous reasons why development falls short of its goal; lack of time or budget, not having the required skill set, the limitations of technology, or most critically, a misunderstanding of the true business need, to name a few.

This misunderstanding can come from any number of sources. Sometimes the business itself does not know its real need or cannot articulate it sufficiently. Important aspects of the need are also often omitted or misconstrued when translated from business to technical terms. Other times, the development team is siloed and does not have the proper context to adequately grasp the business need and deliver accordingly. Most common, though, is that the business attempts to convey the need using traditional communication methods, such as a written specification or a requirements document, which fall short of providing enough detail to fully implement and test the system.

This is significant because recent studies[1] show that over 80% of software defects originate from the requirements elicitation and design stages of the SDLC. One of the primary reasons is that the written English language is inherently ambiguous, even to those with the highest degree of mastery. Many business stakeholders are adept writers, but this method of communication leaves much to be desired. One effective way to circumvent to downfalls of the written word is to augment specs and requirements with pictures that illustrate and help visualize the concept that is being conveyed.

As I mentioned in my abstract, a well-known saying goes: “A picture is worth a thousand words”. While there is some debate as to whether or not this adage holds true universally, I will accept it as a premise for the sake of this work. What kind of pictures, then, are useful when it comes to helping the SDLC achieve what it sets out to do? It turns out that there is no lack of useful visual aids; from simple ‘back of the napkin’ sketches, to abstractions brought to light on a whiteboard, to formal diagrams that have defined standards and best practices. I will focus on the more formal types of pictures in this work; however, it is worth mentioning that less formal types can still be beneficial.

It is important to note that not all written specs are ‘bad’ specs. No one sets out to write a ‘bad’ spec; rather the spec feels accurate to the author who understands the need but fails to convey it to a reader is not already familiar with it. In other words, I could write a requirement or user story that is complete and clear from ***my*** perspective but can be interpreted very differently by someone else. I recently had a conversation with my boss where I said I was having a meeting with a customer *next Thursday* (we had the conversation on a Monday). He asked for clarification on the date. To me, next Thursday obviously meant the Thursday of the following week. He took it to mean the Thursday of the current week, being the next Thursday we would encounter. Even though ***I*** thought I was clear, there was ambiguity in my statement.

There are useful tactics and techniques that can be used to make written specs less ambiguous. Formal structured language, for example, can assign well-defined meanings to words that may other be interpreted multiple ways. In the ‘next Thursday’ example, I could have augmented the description of the day of the meeting with a numerical date. Instead of ‘next Thursday’, I could have said ‘Thursday, July 15th, 2021’. There is only one way to interpret the date in this format.

It is worth mentioning that it is just as possible to create models or diagrams that are ambiguous or confusing as it is to author written specs that are ambiguous or confusing. One should not simply create models or diagrams for the sake of having them without having a strategy or plan in place on how they will be implemented. In order for models or diagrams to add value, they must be clear, well designed, and thought out. Fortunately, many types of diagrams have conventions, best practices, and even restrictions on what you can do in the diagram in order to help ensure the completed work does what it sets out to do.

So, what does it look like to apply the approach of creating models and diagrams (more on this distinction later) to the SDLC? Fortunately, we are able to create models and diagrams at every stage of the SDLC, leveraging the features and benefits of different types of models and diagrams that best suit the various stages. Some types of diagrams can evolve through stages of the SDLC, taking on more content or growing in detail as the project progresses. Shortly, I will discuss these features and benefits during the various stages of the SDLC, including types that can go through this evolution.

Before I discuss the features and benefits, it is important to state that one way to categorize models and diagrams is as such: Behavioral and Structural (also sometimes called dynamic and static respectively). The Object Model Group (OMG) maintains the standard for the Unified Modeling Language (UML) and describes the two categories as follows.

2.1 Behavioral Diagrams

“Behavioral Diagram: emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams, and state machine diagrams.”

2.1.1 Activity Diagrams

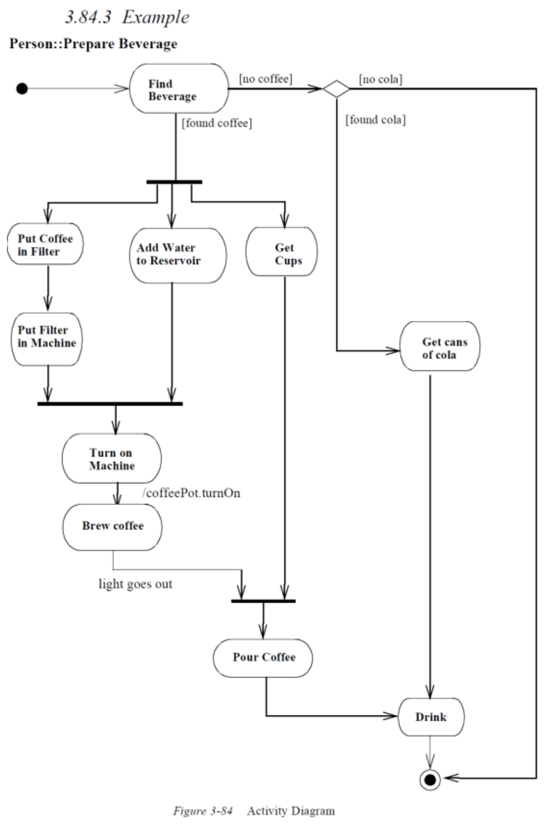


Figure 1: Example Activity Diagram[2]

An example of a Behavioral Diagram is an Activity Diagram. An Activity Diagram, as defined by the OMG, is *"…a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states…The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events)."*

Activity diagrams show activities, nodes, splits and joins, and the 'flow' of the process through these elements. Critical Logic used Activity Diagrams at a major financial institution to help aide their IT department's transition from a Waterfall-oriented development methodology to an Agile development methodology. While the digital transformation had many aspects to it, introducing diagrams into the process streamlined communication between the team members. Business stakeholders were able to create a high-level version of the diagram. We considered this initial version the minimum viable product, or MVP, view of the business need. The MVP version was then handed off to the QA stakeholders who more fully fleshed out the diagram with exceptions and implementation-level details. The implementation version was then provided, along with the user stories and acceptance criteria, to the development team. The result was a common understanding of the business need at every step of the way, along with the following benefits:

1. A reduction in overall solution design and development time
2. Reduced time and cost of test design
3. A higher quality product delivered in each sprint

2.1.2 Use Case Diagrams

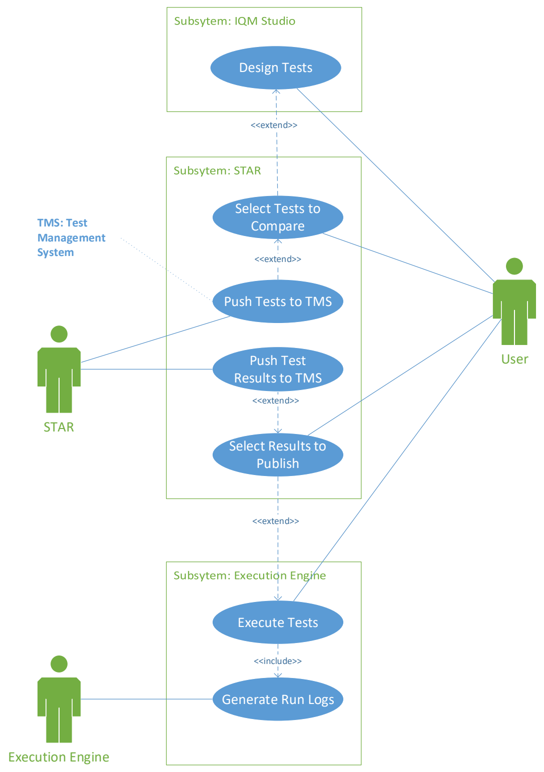


Figure 2: Example Use Case Diagram[3]

Another example of a Behavioral Diagram is a Use Case Diagram. A Use Case Diagram is *"…a graph of actors, a set of use cases, possibly some interfaces, and the relationships between these elements. The relationships are associations between the actors and the use cases, generalizations between the actors, and generalizations, extends, and includes among the use cases. The use cases may optionally be enclosed by a rectangle that represents the boundary of the containing system or classifier."*

Use Case diagrams, sometimes called Function Maps, show the different functions that make up a system as well as the actors that invoke those functions. Notice that each node, or Use Case, begins with a verb. Critical Logic utilizes Use Case Diagrams in its own internal development projects. Use Case Diagrams helped Critical Logic define the current state of the software product, IQM Studio. With a clear picture of their as-is system, Critical Logic was then able to define customer needs and identify functional gaps which could be put on their product roadmap to define their to-be system. In the theme of using diagrams to bring people together, stakeholders from all business areas (including Executives, Product Owners, software QA, and developers) were able to study this roadmap and see how the various functions of IQM Studio related to each other and understand the business value of implementing new functions.

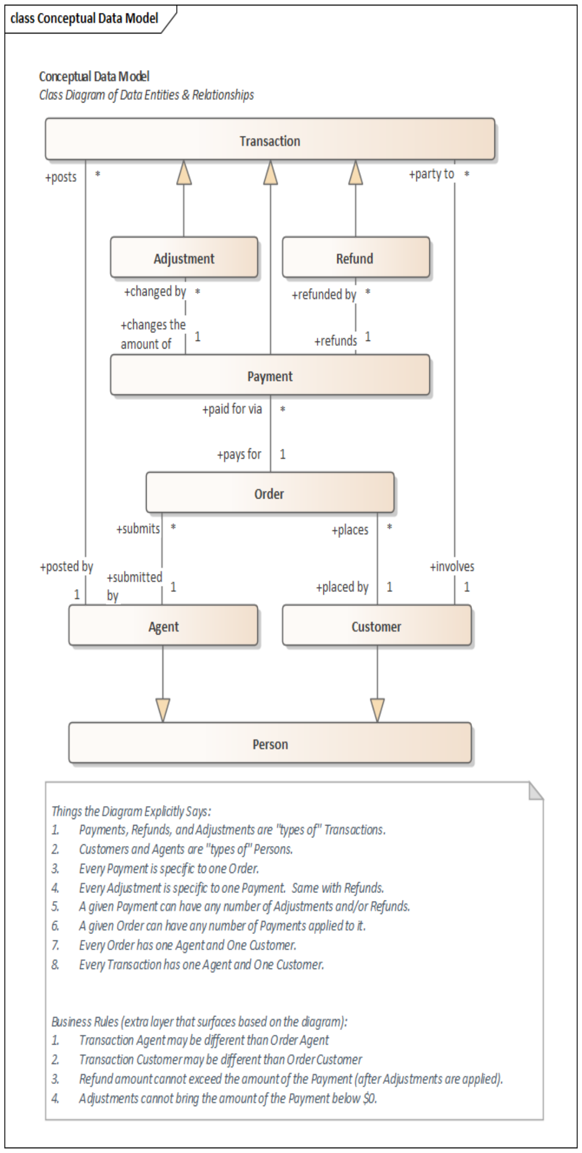
2.2 Structural Diagrams

"Structural Diagram: emphasizes the static structure of the system using objects, attributes, operations and relationships. It includes class diagrams and composite structure diagrams.[4]"

Structural diagrams describe the static, unchanging elements of a system while Behavioral diagrams describe a system 'in action'. When used appropriately, both categories of diagrams serve a useful purpose for describing the business system and the two categories complement each other, meaning that both categories of diagrams can be used in parallel. It should also be noted that while there are currently

14 types of models and diagrams recognized by the OMG UML standard, many other types of diagrams exist that can be useful in their own right.

2.2.1 Class Diagrams



Things the Diagram Explicitly Says:

1. *Payments, Refunds, and Adjustments are "types of" Transactions.*
2. *Customers and Agents are "types of" Persons.*
3. *Every Payment is specific to one Order.*
4. *Every Adjustment is specific to one Payment. Same with Refunds.*
5. *A given Payment can have any number of Adjustments and/or Refunds.*
6. *A given Order can have any number of Payments applied to it.*
7. *Every Order has one Agent and One Customer.*
8. *Every Transaction has one Agent and One Customer.*

Business Rules (extra layer that surfaces based on the diagram):

1. *Transaction Agent may be different than Order Agent*
2. *Transaction Customer may be different than Order Customer*
3. *Refund amount cannot exceed the amount of the Payment (after Adjustments are applied).*
4. *Adjustments cannot bring the amount of the Payment below $0.*

Figure 3: Example Class Diagram 1

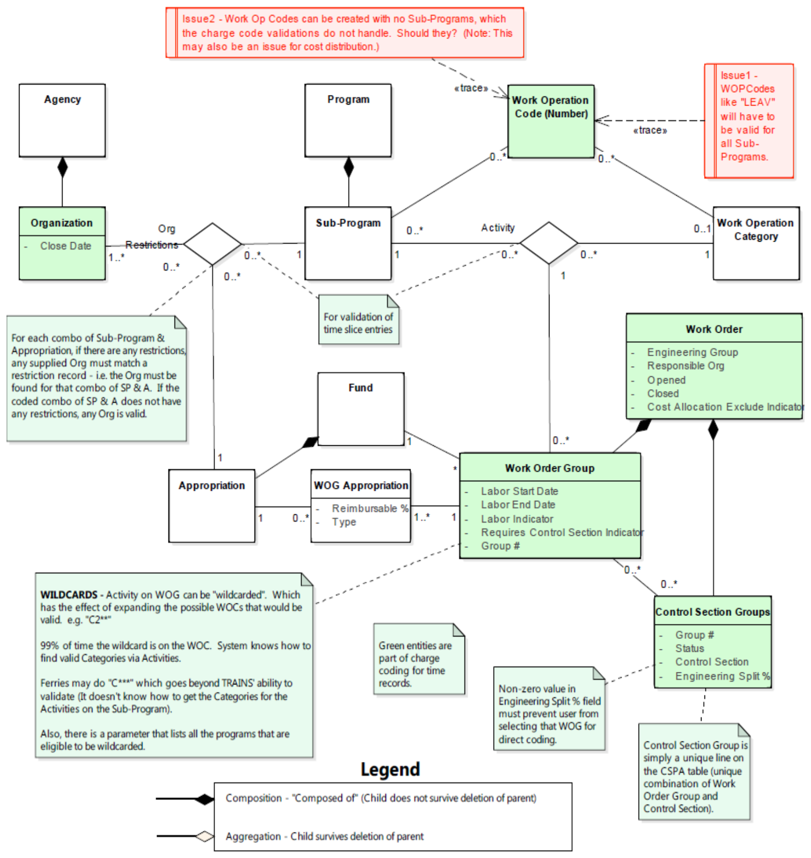
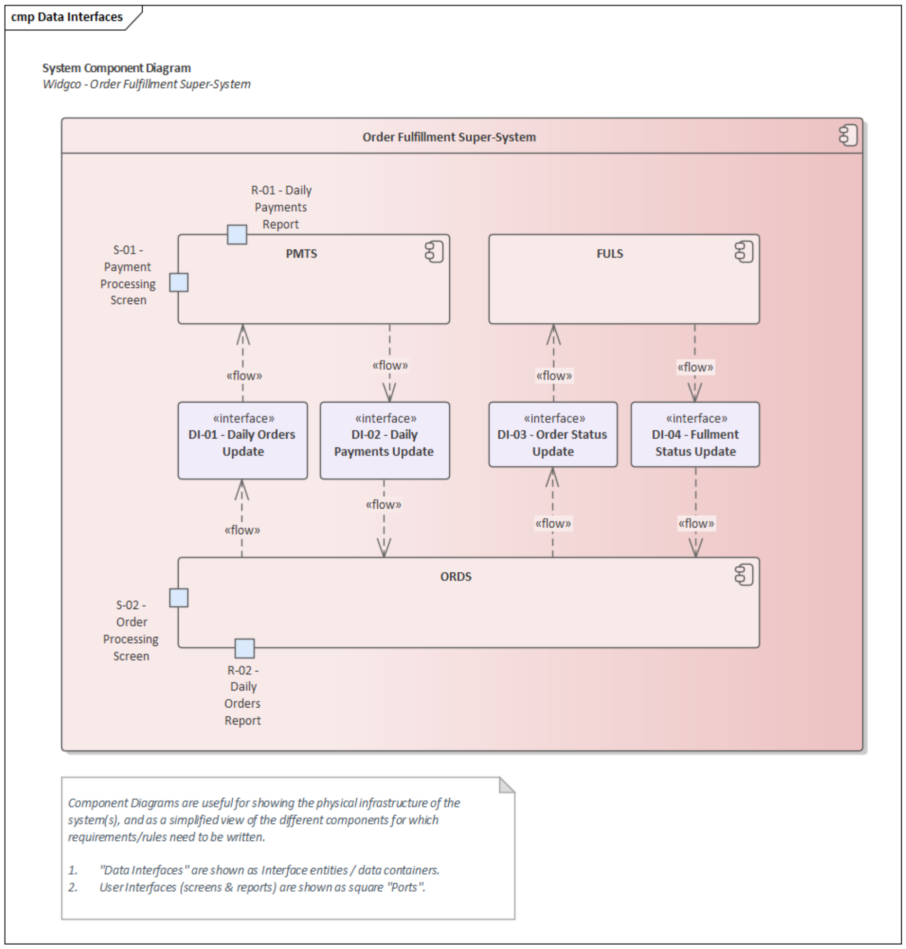

Issue2 - Work Op Codes can be created with no Sub-Programs, which the charge code validations do not handle. Should they? (Note: This may also be an issue for cost distribution.)
«trace»
Agency
Program
Work Operation Code (Number)
«trace»
Issue1 - WOPCodes like "LEAV" will have to be valid for all Sub-Programs.
Organization
- Close Date
Org Restrictions
Sub-Program
Activity
Work Operation Category
0..*
0..1
1..*
1
For validation of time slice entries
For each combo of Sub-Program & Appropriation, if there are any restrictions, any supplied Org must match a restriction record - i.e. the Org must be found for that combo of SP & A. If the coded combo of SP & A does not have any restrictions, any Org is valid.
Work Order
- Engineering Group
- Responsible Org
- Opened
- Closed
- Cost Allocation Exclude Indicator
Fund
Appropriation
WOG Appropriation
- Reimbursable %
- Type
Work Order Group
- Labor Start Date
- Labor End Date
- Labor Indicator
- Requires Control Section Indicator
- Group #
Control Section Groups
- Group #
- Status
- Control Section
- Engineering Split %
WILDCARDS - Activity on WOG can be "wildcarded". Which has the effect of expanding the possible WOCs that would be valid. e.g. "C2**"
99% of time the wildcard is on the WOC. System knows how to find valid Categories via Activities.
Ferries may do "C***" which goes beyond TRAINS' ability to validate (It doesn't know how to get the Categories for the Activities on the Sub-Program).
Also, there is a parameter that lists all the programs that are eligible to be wildcarded.
Green entities are part of charge coding for time records.
Non-zero value in Engineering Split % field must prevent user from selecting that WOG for direct coding.
Control Section Group is simply a unique line on the CSPA table (unique combination of Work Order Group and Control Section).
Legend
Composition - "Composed of" (Child does not survive deletion of parent)
Aggregation - Child survives deletion of parent

Figure 4: Example Class Diagram 2[5]

An example of a Structural Diagram is a Class Diagram. A Class Diagram is *"…a collection of static declarative model elements, such as classes, interfaces, and their relationships, connected as a graph to each other and to their contents."*

Class Diagrams show various levels of detail of the system they represent. The first example above shows only the classes and their relationships, while the second example includes attributes about the classes. Class diagrams are often thought of as technical diagrams used only by Object-Orient programmers. Critical Logic has used Class Diagrams as a powerful tool for bridging the gap between business and technical stakeholders through an abstraction layer. Class Diagrams also provide the vocabulary for rule and requirement writing, again allowing different stakeholders to communicate using the same language.

2.2.2 Component Diagrams


cmp Data Interfaces
System Component Diagram
Widgco - Order Fulfillment Super-System
Order Fulfillment Super-System
R-01 - Daily Payments Report
S-01 - Payment Processing Screen
PMTS
FULS
«flow»
«flow»
«flow»
«flow»
«interface» DI-01 - Daily Orders Update
«interface» DI-02 - Daily Payments Update
«interface» DI-03 - Order Status Update
«interface» DI-04 - Fullment Status Update
«flow»
«flow»
«flow»
«flow»
ORDS
S-02 - Order Processing Screen
R-02 - Daily Orders Report

Component Diagrams are useful for showing the physical infrastructure of the system(s), and as a simplified view of the different components for which requirements/rules need to be written.

1. *"Data Interfaces" are shown as Interface entities / data containers.*
2. *User Interfaces (screens & reports) are shown as square "Ports".*

Figure 5: Example Component Diagram

Another example of a Structural Diagram is a Component Diagram. A Component Diagram is *"…a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships."*

Like Class diagrams, Component Diagrams are often thought of as technical diagrams used only by Object-Orient programmers. They also serve as a powerful tool for bridging the gap between business and technical stakeholders through an abstraction layer. Critical Logic has developed Component Diagrams on customer projects to bring attention to the various interfaces of a system or systems that are part of the implementation solution. Component Diagrams can be effective for developing a holistic view of the solution as well as everything the solution touches.

2.3 Uses of diagrams during the SDLC

There are no hard and fast rules for which models and diagrams should be created during the various stages of the SDLC, though some do tend to lend themselves to be most effective at certain times. One approach to selecting a type of diagram is to create them that are strategic in nature early on in the process and, as the process advances, transition to models and diagrams that are more functional in nature. For example, while in the planning and analysis phases, creating diagrams regarding organization structure (Organizational Charts), As-is and To-be processes (Business Process Models), business activities (Activity Diagrams), or associations (Entity Relationship Diagrams) could be quite useful. As the project advances to the design, development, testing, and implementation phases, creating models and diagrams regarding specific functions (Use Case Diagrams, sometimes referred to as Function Maps), system architecture (Database Diagrams, Component Diagrams), intended system behavior (Cause-Effect Models, Activity Diagrams), or system transitions (State Machine Diagrams) will greatly augment the text-based definitions of the system.

Notice that Activity Diagrams are effective to create in both the early stages of the SDLC as well as the latter. As mentioned previously, some types of diagrams lend themselves well to evolving through the SDLC. Cause-Effect Models are another type that can evolve. Models and diagrams that can evolve throughout the process are especially valuable because they continue to be useful as the process advances from one group of stakeholders to the next, adding a level of continuity that could otherwise be lost.

There are benefits to creating diagrams in both pre-development and post-development stages of the SDLC. When creating models early on, some of the benefits include:

- Developing a clear understanding of the need to be met
- Accelerating the design stage
- Allowing questions to be brought to light and discussed by the appropriate stakeholders
- Unveiling ambiguity that may exist in the business requirements
- Reducing maintenance burden in the Maintenance and Operation stages

Creating models and diagrams helps stakeholders visualize the need in a way that can be hard to do in pure text-based descriptions. Similarly, when designing the ‘what’ part of the way the need will be met (in a solution-independent fashion), creating models and diagrams reinforce that what will be built ***should*** be built. For example, a Use Case Diagram that shows a complete, non-overlapping list of all functions the new system will contain will much more effectively and efficiently be shared among the vested stakeholders than a text-based document on its own. This in turn will help the design come together more efficiently with less back-and-forth between stakeholders. This greatly reduces the length of the design phase. Models and diagrams hold a significant advantage when it comes to bringing ambiguity regarding the intended behavior of a system to light. The ambiguity can then be discussed and driven to resolution in the early stages of the SDLC, saving cost and rework from occurring later on.

Creating and maintaining models and diagrams in the later stages of the SDLC also provides number benefits:

- Provides a clear sense of how the solution will be implemented
- Unveils ambiguity that was not discovered during initial design phase
- Clarifies confusion
- Fills in the gaps in the spec
- Makes the massive manageable
- Creates a simplified view of the complex
- Can be fleshed out to contain very deep level of detail (e.g., equivalence classes, boundary values, combinations)
- Some types allow for tests to be derived from the diagram – others can even automatically generate tests from the diagrams

When the business system being developed has a graphical user interface, or GUI, models and diagrams are effective to convey what the GUI should look like. For example, imagine how much effort it would take to describe what a website’s homepage should look like without using models or diagrams (such as mockups or wireframes). Even when the system does not have a GUI or there are GUI-independent components to the system, models and diagrams are still useful to describe how these components should work. Using a Database diagram to represent the architecture of a backend database is significantly more efficient than an attempt to describe the architecture using only words.

As with creating models and diagrams in the early stages of the SDLC, creating them during the later stages helps to bring to light questions, ambiguities, and gaps in the specifications, again saving cost and preventing rework later on. This also is an effective way to eliminate defects from being introduced into the system due to unclear specs.

Models and diagrams also serve a helpful purpose when it comes to defining scope. Certain types of diagrams, Use Case Diagrams for example, can be used to fully define the functions of a system, making what seems like a massive, complex interaction of disparate things seem manageable and understandable.

When models and diagrams are leveraged to capture low levels of detail, they serve multiple purposes. For example, when Cause Effect Diagrams are used to define the intended behavior of a system, they can take on increasing levels of detail as a project moves from design to development to testing and, finally, the implementation stage. A Cause Effect Diagram can start out defining the 'go-right' behaviors of a system early on and then have more details, such as exceptions or alternate paths added as they are defined. Cause Effect Diagrams are also excellent for defining boundary value conditions, equivalence classes, and logical combinations. Certain types of Cause Effect Diagrams can even automatically generate high coverage test cases from the information in the diagram, an effective type of Model Based Testing (MBT).

A major benefit of MBT is that the models themselves are multi-use artifacts. Many other artifacts developed during the SDLC are single-use. Functional specifications, for example, serve the sole purpose of specifying how the solution will be implemented. While they can serve as a guide to developing other artifacts, it still requires an additional independent step to generate the additional artifacts.

Models used in MBT, however, are different. They serve the valuable initial purpose in bringing stakeholders of various levels together to give them a communication conduit that develops a common understanding of the business need and solution to be implemented. These models can go on to serve an additional purpose when they are used to create tests from the information defined in the models. Some commercially available tools can even *automatically generate* tests from the model. This allows the model to continue to add value throughout the stages of the SDLC, including into the maintenance and operation phases. The next section will expand on the benefit recognized by implementing MBT.

3 Benefits of Moving to Model Based Testing

Model Based Testing has been increasing in popularity, but it is far from a new concept. In 2008 a Department of Defense contractor, General Dynamics, was challenged by the United States Navy to reduce testing costs while at the same time increase the quality of their mission- and life-critical systems, which was no small feat[6]. A year later, they were introduced to an MBT solution that they implemented for their Navy project. In addition to saving General Dynamics $3 Million on that project alone, Ron Townsen, the Sr. Lead Engineer at General Dynamics had this to say:

"[DTT, the modeling tool], along with the higher quality of model-based testing, gave us the approach we have been looking for...Especially in areas of complex design and safety critical needs"

General Dynamics is not alone in their implementation of MBT. The Model-Based Testing User Survey has been administered and published by a number of individuals since at least 2011. The most recent survey was administered and published in 2019 by two members of the German Testing Board, Anne Kramer and Bruno Legeard. One question on the survey asked respondents about their expectations for implementing MBT at their organization, to which over 60% of respondents answered, "We wish to improve the communication between stakeholders." The survey goes on to ask, "From your current experience, does MBT fulfill those expectations?". A staggering 75% of respondents answered "Yes" (46%) or "Partly" (29%) to this question.[7]

Studying additional questions from the survey reinforces the idea that models and diagrams can be used at any stage of the SDLC, requirements elicitation for example, to great benefit. These questions also demonstrate how models and diagrams evolve from when they are built as they progress from stage to stage.

Another excellent example of this is Sun Microsystems, which turned to MBT to help with their unique challenges after being acquired by Oracle in 2010[8]. At the time, Sun's 'Configurator' was a system designed to give their sales force a current view of their product offering, ensuring that the most current and compatible options were offered. Given the rapidly changing state of the complex hardware and software landscape, the Configurator's nearly 60,000 business rules were constantly in a state of flux. In a given month, an average of 25% of the rules would undergo some form of change. By representing these rules in Cause Effect Diagrams, then automatically generating tests from the diagrams, several significant benefits were realized.

First, the rate of release for the configurator doubled, increasing from once every other month to once a month. This means that the Configurator was always returning the most up to date offering of Sun's product line. It also ensured that quotes are accurate and timely.

Second, several consecutive zero-defect releases went into production. This was something that Sun had not been able to accomplish prior to implementing MBT with Cause Effect Diagrams. These zero-defect releases give their sales team confidence that the information they are giving their customers is correct and reliable.

Third, the amount of testing resources to accomplish these achievements actually ***decreased by a third***. By freeing up testing resources from manual testing, Sun's SQA teams are able to focus on value-added QA activities. Sun also reduced their head count of testing resources, allowing these resources to be moved to other areas such as to the dev team.

Salesforce is another household name that has benefited significantly from implementing MBT[9]. Straining from rapid growth due to their successful platform and services, Salesforce.com had no consistent process for capturing requirements during the elicitation stage. This led, unsurprisingly, to missed or incorrectly implemented requirements causing defects in production. Given Salesforce's reputation as a world leader in Software as a Service, this was not sustainable. As mentioned above, the requirement authors at Salesforce.com set out to create complete, unambiguous, and implementable requirements. In their case, and many cases just like this, it was a limitation of resources available that sabotaged their ability to achieve their goal.

Within weeks of implementing Model Based Testing and focusing on visualizing requirements for clarity and completeness, Russ Nelson, the Vice President of Applications Development had this to say:

"[The MBT implementor] reduced our requirements elicitation investment by 50% while dramatically increasing the quality of the software we deliver…[and] they allow us to leverage our existing staff 4:1."

What does the process of creating Cause Effect Models which automatically generate tests look like in practice then? Let's walk through it.

3.1 Model Based Testing in Practice

The first step in a disciplined and effective approach to modeling requirements is to perform an initial analysis of the requirement artifacts that have been developed so far. To illustrate this, I will describe another real-world application. Critical Logic engaged in a project in 2015 – 2016 to assist in the modernization of a Veteran's Affairs (VA) system that would centralize and make nationally available medical records of US Armed Forces Veterans, specifically their medications and adverse reactions to medications. By the time Critical Logic engaged in the project, all of the business and functional requirements had been defined by the VA business sponsor and turned over to the vendor for implementation.

After winning the contract and taking delivery of the requirements, the vendor quickly realized that the 'complete' requirements were anything but. This was a good realization, but they were left without a clear path forward to develop the solution the VA would accept. The requirements were simply not complete and too ambiguous for development and implementation. Due to the nature of the contract, requesting new requirements was also not an option. They needed an approach that would allow them to leverage the work the VA had done AND result in a complete requirement set, which is when Critical Logic became involved.

Critical Logic's first task was to represent the requirements in Use Case Diagrams. Doing so demonstrated the areas that the VA requirement authors had failed to define. The Use Case Diagrams proved effective at showing exactly where additional requirement elicitation or clarification was required. This approach allowed the vendor to go back to the VA in a manner that showed they were leveraging all of the work the VA had undertaken in authoring the requirements, while asking only targeted questions to fill the requirement gaps and clarify ambiguities.

Once a *truly* complete set of requirements had been defined, Critical Logic began representing the requirements in Cause Effect Diagrams. Cause Effects Diagrams created at this stage built upon the work done in the Use Case Diagrams and included functional requirement-level information that unearthed additional requirement gaps and ambiguities. The image below is an example of a Cause Effect Diagram from the project that unearthed functionality that was not defined well enough to be implemented by the development team.

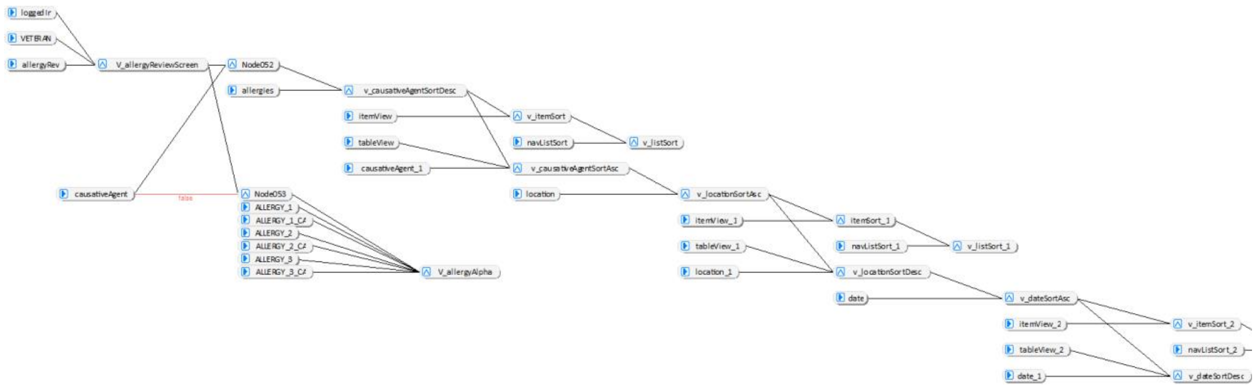
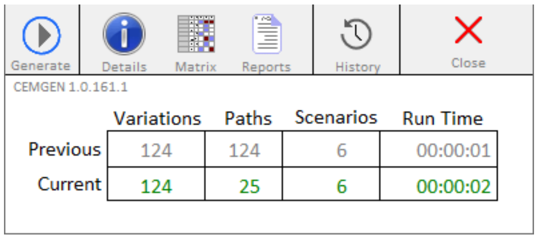


Figure 6: Real-world Cause Effect Diagram

In general, each function defined in the Use Case Diagrams corresponded to a set of functional Cause Effect Diagrams. Once a Cause Effect Diagram was created and taken to a point where it was ready for review, Critical Logic would review the diagram with the VA and Vendor subject matter experts (SMEs) in ambiguity review sessions, bringing to light any questions that arose during the diagram creation process. When questions could not be answer in real-time, Critical Logic would track the ambiguity and ensure that it was driven to resolution.

Once the diagrams had resolved all of the requirement gaps and ambiguities, a test generation algorithm was invoked that took the information in the Cause Effect Diagram as input and automatically generated tests from the model.



CEMGEN 1.0.161.1

	Variations	Paths	Scenarios	Run Time
Previous	124	124	6	00:00:01
Current	124	25	6	00:00:02

Figure 7: The test generation algorithm created 6 scenarios from the diagram

Ideally at this point, the tests would be reviewed by the SMEs for sign off. Because the tests had been generated from the model, and because the model had been reviewed by the VA and Vendor SMEs, the test review process was incredibly efficient, and signoff was given with minimal updates needed.

The benefits of using Use Case and Cause Effect Diagrams to augment and enhance the VA-authored requirements were apparent in the first sprint retrospective where the business was able to see the MVP. Sprint after sprint, as the VA and vendor teams embraced the CEM process and the ambiguity and test review sessions, efficiency increased, and the development team consistently delivered nearly defect-free product that truly represented the business needs. Ultimately, the product was accepted, and the implementation was successful.

In each of these cases, we saw organizations with different challenges and goals; all of which were realized with the assistance of MBT. General Dynamics was able to improve the quality of their mission- and life-critical systems while at the same time reducing their testing costs. Sun Microsystems were able to keep up with the frenetic rate of change required for them to maintain their market-leading status. Salesforce.com saw that they could improve and standardize their requirements elicitation process while at the same time allowing their staff to be more efficient with their time. The VA implementation vendor took the incomplete, ambiguous requirements authored by their customer and delivered a truly complete set of requirements and a product that fully represented these requirements.

In each example, creating models provided the project artifacts that allowed stakeholders to come together and enhance the quality of the system in development. With General Dynamics and the VA vendor, the models provided a graphical representation that was easily consumable by the business and dev teams. By discussing questions in the context of the graphical representation of the requirements, the gap between the business and development stakeholders was bridged. With Salesforce.com, CEMs were used to enhance the elicitation process, putting the end users and business analysts on the same page. At Sun, CEMs brought stakeholders together to allow their business and development teams to keep up with the required rapid pace of change that Sun needed to maintain their market leader status.

4 Moving from Model Based Testing to Automated Testing

Once an organization has progressed to the point where they are realizing the benefits of Model Based Testing, there is an additional level that can be achieved to further promote unity between all stakeholders. Before I get into the details of this final step, however, let's discuss the distinction between a model and a diagram.

A model is an abstract representation of something, a system for example, where every part of that system is defined exactly once. A diagram is an instance of the model. Each element that makes up that system is referenced when that element is needed to define the system in a diagram. Each time an element is used, it is a *reference* to the element. This eliminates the possibility of having the same element described in more than one way and ensures that each element referenced is exactly the same each time it is used.

A diagram can be thought of as a specific view of the model. Different elements from the model can be assembled in a way to create the view you are interested in, and the same elements from the model can be used to create any number of different diagrams.

This distinction between models and diagrams is important when talking about automated testing because there is an incredibly effective way to leverage models, MBT, and automated testing frameworks that takes advantage of the concept of a model. This concept was put into practice at General Dynamics, as mentioned above. Specifically, the teams at General Dynamics created Cause Effect Models to represent the two major components of their integration project. These two systems combined had over 2,500 requirements to define their intended behaviors. Even the most skilled analyst could never ensure that this was a complete, unambiguous, non-contradictory list of statements using only formal structured text. By representing these requirements in a disciplined, structured fashion using Cause Effect Models, the teams were able to realize all of the benefits mentioned above. They raised questions via ambiguity review sessions in a regular cadence and were able to get clarification and improve the requirements before they were approved and sent to the development team. Eventually, the development team started sending a representative to participate in the ambiguity review sessions to get a firsthand account of the clarification process. In this instance, modeling helped bring stakeholders together in a situation where they otherwise would have remained siloed.

Once the Cause Effect Models were created, the team was able to automatically generate a full coverage set of tests from the models. Because the Cause Effect Models were created in the design stage, and the tests were generated from the models, the tests were developed much sooner in the SDLC than typical. This meant that the tests were able to be sent to development team along with the requirements. The benefit of this was that the development team knew exactly what tests would be executed in order to validate the system behavior, giving them an additional design artifact to base their development on.

The General Dynamics team did not stop there, though. They then created an automation framework that was exposed at the model level. Doing so allowed them to build an automation test script library that correlated perfectly to the tests generated from the models by treating the automation framework like a model - each object in the system under test was defined once and then called each time it was a part of a test. These objects and their applicable actions were mapped to the steps defined in the models. After these objects and actions were mapped, or scripted as it is often referred to as, the tests automatically generated from the models also automatically generated the automation scripts.

To illustrate this process, here is a generic example that shows an example of a Cause Effect Model that is used to generate tests, and the corresponding scripting that allows the model to also generate automation scripts. This generic example represents a Library Information System, or LIS.

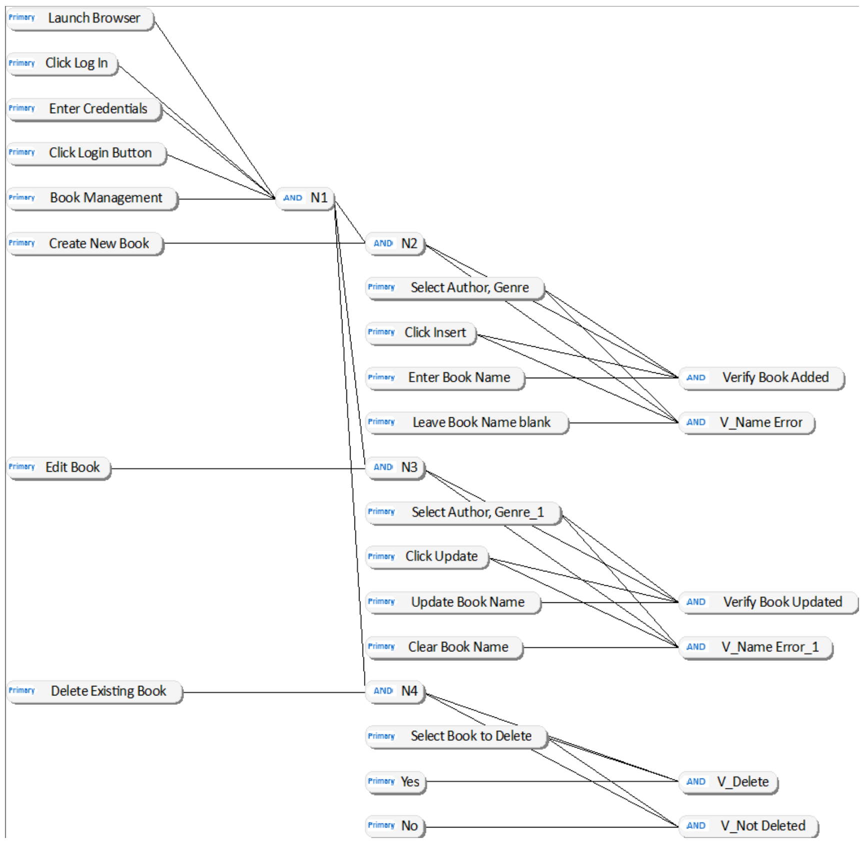
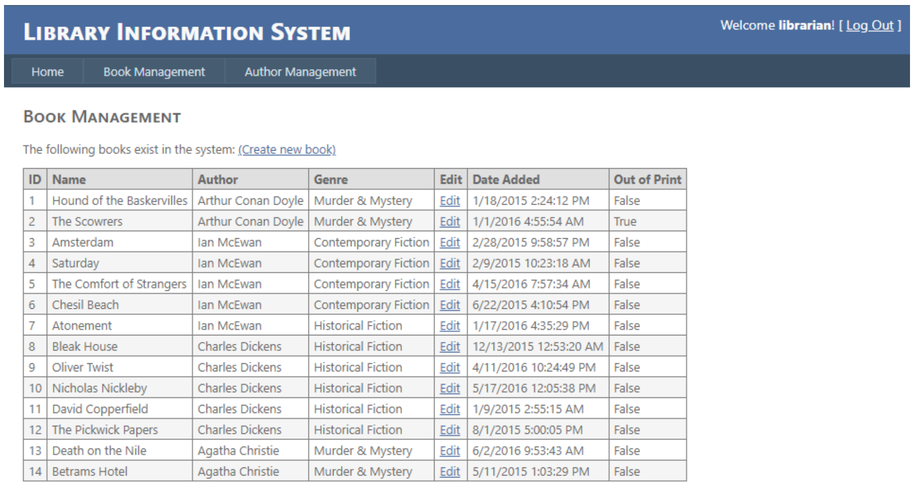


Figure 8: Cause Effect Model representing the LIS requirements

Here's what the LIS looks like:



LIBRARY INFORMATION SYSTEM

Welcome **librarian**! [Log Out]

Home | Book Management | Author Management

BOOK MANAGEMENT

The following books exist in the system: (Create new book)

ID	Name	Author	Genre	Edit	Date Added	Out of Print
1	Hound of the Baskervilles	Arthur Conan Doyle	Murder & Mystery	Edit	1/18/2015 2:24:12 PM	False
2	The Scowrers	Arthur Conan Doyle	Murder & Mystery	Edit	1/1/2016 4:55:54 AM	True
3	Amsterdam	Ian McEwan	Contemporary Fiction	Edit	2/28/2015 9:58:57 PM	False
4	Saturday	Ian McEwan	Contemporary Fiction	Edit	2/9/2015 10:23:18 AM	False
5	The Comfort of Strangers	Ian McEwan	Contemporary Fiction	Edit	4/15/2016 7:57:34 AM	False
6	Chesil Beach	Ian McEwan	Contemporary Fiction	Edit	6/22/2015 4:10:54 PM	False
7	Atonement	Ian McEwan	Historical Fiction	Edit	1/17/2016 4:35:29 PM	False
8	Bleak House	Charles Dickens	Historical Fiction	Edit	12/13/2015 12:53:20 AM	False
9	Oliver Twist	Charles Dickens	Historical Fiction	Edit	4/11/2016 10:24:49 PM	False
10	Nicholas Nickleby	Charles Dickens	Historical Fiction	Edit	5/17/2016 12:05:38 PM	False
11	David Copperfield	Charles Dickens	Historical Fiction	Edit	1/9/2015 2:55:15 AM	False
12	The Pickwick Papers	Charles Dickens	Historical Fiction	Edit	8/1/2015 5:00:05 PM	False
13	Death on the Nile	Agatha Christie	Murder & Mystery	Edit	6/2/2016 9:53:43 AM	False
14	Betrams Hotel	Agatha Christie	Murder & Mystery	Edit	5/11/2015 1:03:29 PM	False

Figure 9: The LIS sample application[10]

Once a system such as the LIS is in production, an automation engineer can develop an automated testing framework of the system, mapping out all of the objects and assigning the valid actions that can be taken against those objects. For example, the user is able to click the 'Edit' link for a book in the LIS table. In this case, the 'Edit' link is the object and 'click' is the action. (Specifically, left-mouse button clicking is the action.) If we look at a subset of the CEM diagram, we see that the model represents the 'Create New Book' function:

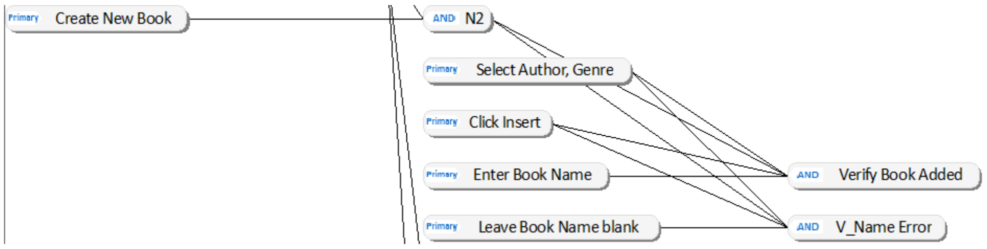
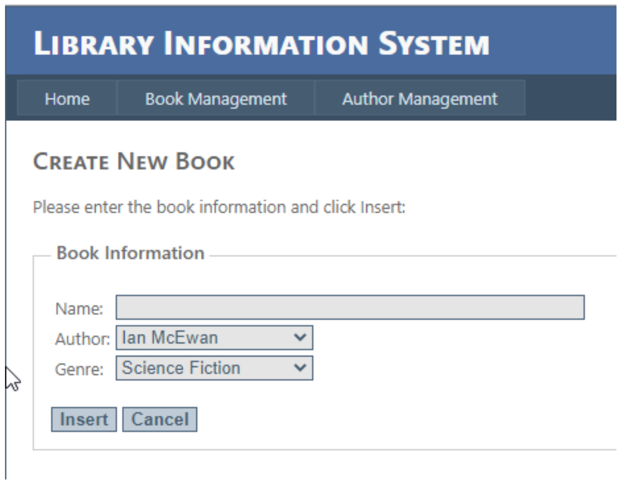


Figure 10: Subset of the LIS CEM

In the LIS itself, this is what the 'Create New Book' function looks like:



LIBRARY INFORMATION SYSTEM

Home | Book Management | Author Management

CREATE NEW BOOK

Please enter the book information and click Insert:

Book Information

Name:	
Author:	Ian McEwan
Genre:	Science Fiction

Figure 11: Create New Book function of LIS

The objects we are interested in for the sake of creating automated tests are Name (text field), Author (drop down box), Genre (drop down box), Insert (button), and Cancel (button). Each of these objects have different actions that can be taken against them. Take the "Name" field for example. Some of the testing-related actions we could take against the "Name" field are: Input text, Input text without clearing existing text, Verify it is enabled, and Verify it contains a certain value. Critical Logic's commercially available tool, IQM Studio allows access to the automation framework with a simple drop-down selector tool that is human readable. In other words, non-technical stakeholders who create or interact with CEMs are able to 'script' the models by selecting these objects and actions. This is what it looks like in the tool:

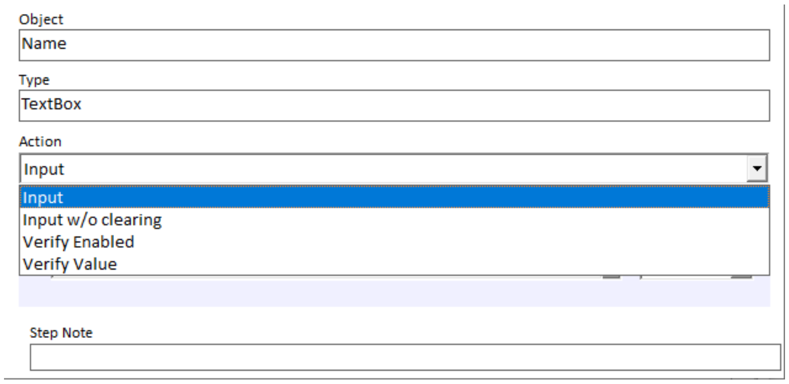
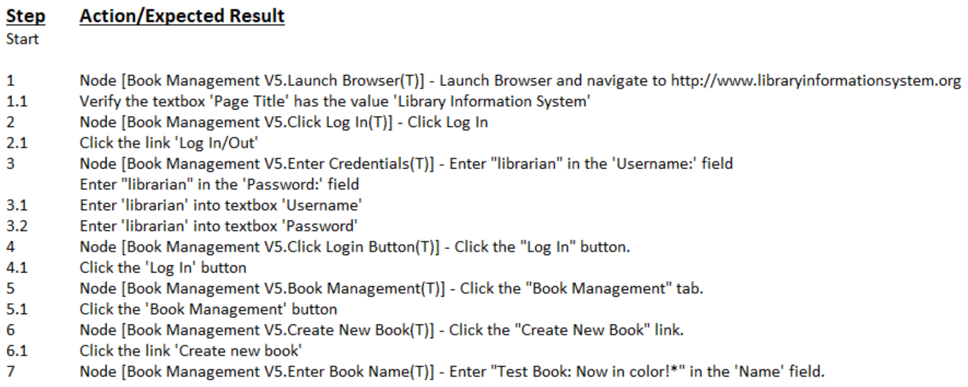

Object: Name
Type: TextBox
Action: Input

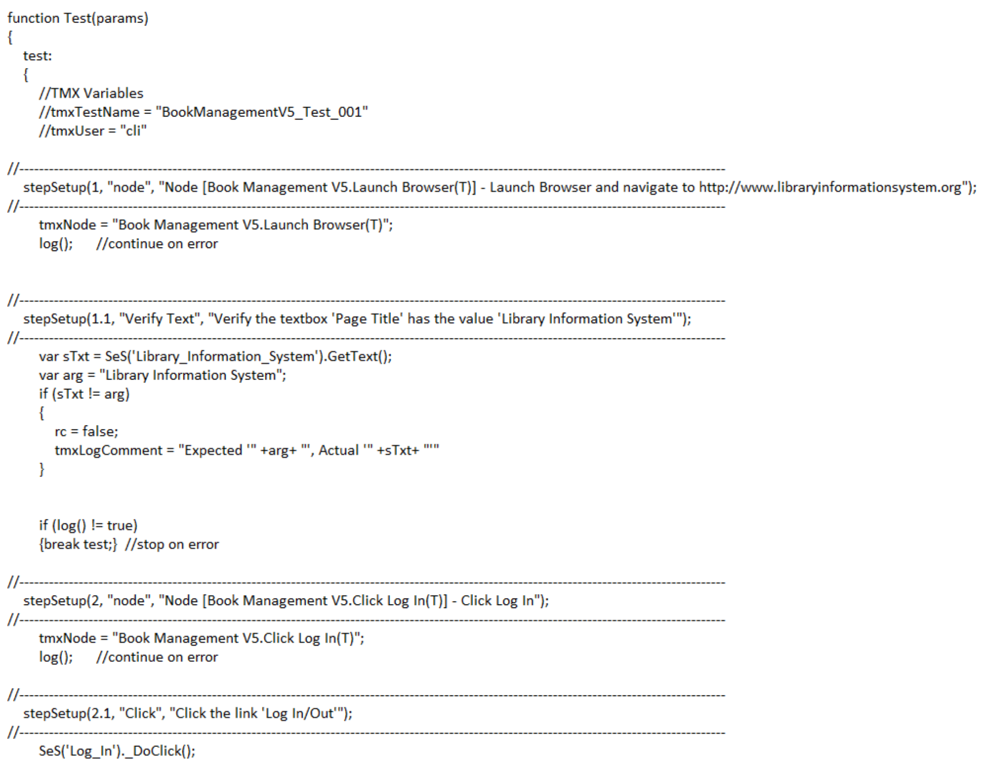
Figure 12: IQM Studio scripting and object and actions

All the modeler has to know is which object is being represented in the model and then choose which action is being taken against that object. Behind the scenes, IQM Studio is linking the scripted steps (the object/action pairs) to the code in the automated testing framework developed by the automation engineer. This splits the automation tester role into two: the automation engineer and the scripter. The automation engineer is responsible for developing and maintaining the automation framework code; that is, the code needed for the automation tool to take actions against the defined objects. The scripter is responsible for associating the proper objects and actions to the correct steps in the CEM. By doing so, IQM Studio's test generation algorithm automatically generates tests ***and*** automated test scripts at the same time!



Step	Action/Expected Result
Start	
1	Node [Book Management V5.Launch Browser(T)] - Launch Browser and navigate to http://www.libraryinformationsystem.org
1.1	Verify the textbox 'Page Title' has the value 'Library Information System'
2	Node [Book Management V5.Click Log In(T)] - Click Log In
2.1	Click the link 'Log In/Out'
3	Node [Book Management V5.Enter Credentials(T)] - Enter "librarian" in the 'Username:' field Enter "librarian" in the 'Password:' field
3.1	Enter 'librarian' into textbox 'Username'
3.2	Enter 'librarian' into textbox 'Password'
4	Node [Book Management V5.Click Login Button(T)] - Click the "Log In" button.
4.1	Click the 'Log In' button
5	Node [Book Management V5.Book Management(T)] - Click the "Book Management" tab.
5.1	Click the 'Book Management' button
6	Node [Book Management V5.Create New Book(T)] - Click the "Create New Book" link.
6.1	Click the link 'Create new book'
7	Node [Book Management V5.Enter Book Name(T)] - Enter "Test Book: Now in color!*" in the 'Name' field.

Figure 13: Example model generated test



```
function Test(params)
{
   test:
   {
      //TMX Variables
      //tmxTestName = "BookManagementV5_Test_001"
      //tmxUser = "cli"

//--------------------------------------------------------------------------------------------------------------------------
   stepSetup(1, "node", "Node [Book Management V5.Launch Browser(T)] - Launch Browser and navigate to http://www.libraryinformationsystem.org");
//--------------------------------------------------------------------------------------------------------------------------
      tmxNode = "Book Management V5.Launch Browser(T)";
      log();     //continue on error


//--------------------------------------------------------------------------------------------------------------------------
   stepSetup(1.1, "Verify Text", "Verify the textbox 'Page Title' has the value 'Library Information System'");
//--------------------------------------------------------------------------------------------------------------------------
      var sTxt = SeS('Library_Information_System').GetText();
      var arg = "Library Information System";
      if (sTxt != arg)
      {
         rc = false;
         tmxLogComment = "Expected '" +arg+ "', Actual '" +sTxt+ "'"
      }


      if (log() != true)
      {break test;}  //stop on error

//--------------------------------------------------------------------------------------------------------------------------
   stepSetup(2, "node", "Node [Book Management V5.Click Log In(T)] - Click Log In");
//--------------------------------------------------------------------------------------------------------------------------
      tmxNode = "Book Management V5.Click Log In(T)";
      log();     //continue on error

//--------------------------------------------------------------------------------------------------------------------------
   stepSetup(2.1, "Click", "Click the link 'Log In/Out'");
//--------------------------------------------------------------------------------------------------------------------------
      SeS('Log_In')._DoClick();
```

Figure 14: Example automation script corresponding to manual test

What this means is that the testers (who are also the modelers and scripters) don't need to know how code and the automation engineer doesn't have to design, author, or maintain tests – only the framework itself!

5 Conclusion

The simple act of drawing a picture can convey so much meaning, information, and context. Creating models and diagrams to augment text-based definitions of a business system's intended behavior adds value at every stage of the SDLC. One of the primary ways models and diagrams do this is by creating a conduit of communication between stakeholder groups who do not always speak the same language. The models and diagrams allow stakeholders of varying degrees of business and technical expertise to have something to point to in order to help them convey concepts and ideas.

More mature organizations can leverage certain types of models and diagrams that allow them to implement Model Based Testing. Because models and diagrams can be created as early as the design phase, MBT can provide the benefit of developing tests much earlier in the SDLC than is typically realized. This provides an additional, valuable design artifact that can be supplied to the development team to provide an extra level of description into the intended behavior of the system being developed.

Finally, organizations can further leverage the models to build an automation framework that has the primary characteristics of a model. This allows teams to access the automation framework at the model level and generate automated testing scripts that can be executed once the business system has been deployed. This means that all people, stakeholders at *any* level, can take control of the quality of their organization's business systems, and the models are a key artifact in enabling them to be able to do so.

6 Selected Glossary of Terms (as defined by the OMG)

1. **State Machine:** The State Machine package is a subpackage of the Behavioral Elements package. It specifies a set of concepts that can be used for modeling discrete behavior through finite state-transition systems…State machines can be used to specify behavior of various elements that are being modeled. For example, they can be used to model the behavior of individual entities (such as, class instances) or to define the interactions (such as, collaborations) between entities. In addition, the state machine formalism provides the semantic foundation for activity graphs. This means that activity graphs are simply a special form of state machines.
2. **Activity Diagram:** A special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states…The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events). In the context of this paper, workflow diagram or swim lane diagram are not regarded as Activity Diagram, and they are out of scope of this paper.
3. **Activity Graph:** The Activity Graphs package defines an extended view of the State Machine package. State machines and activity graphs are both essentially state transition systems, and share many metamodel elements. An activity graph is a special case of a state machine that is used to model processes involving one or more classifiers. Its primary focus is on the sequence and conditions for the actions that are taken, rather than on which classifiers perform those actions. Most of the states in such a graph are action states that represent atomic actions; that is, states that invoke actions and then wait for their completion. Transitions into action states are triggered by events, which can be:
    a. the completion of a previous action state (completion events),
    b. the availability of an object in a certain state
    c. the occurrence of a signal, or
    d. the satisfaction of some condition.

References

1. https://www.academia.edu/2182202/The_pattern_of_software_defects_spanning_across_size_complexity
2. https://www.omg.org/spec/UML/1.4/PDF
3. Copyright, Critical Logic, 2021
4. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/behavior-vs-structural-diagram/
5. Copyright, Critical Logic, 2021
6. https://www.critical-logic.com/assets/General-Dynamics-Case-Study-V5.pdf
7. https://www.cftl.fr/wp-content/uploads/2020/02/2019-MBT-User-Survey-Results.pdf
8. https://www.critical-logic.com/assets/Critical-Logic-SunMicroSystems-CaseStudy.pdf
9. https://www.critical-logic.com/assets/Critical-Logic-SFDC-Case-Study.pdf
10. http://www.libraryinformationsystem.org/Books.aspx