Abstract

Many test efforts depend on scenarios that represent real sequences of transactions and events. Scenarios are important tools for finding problems that matter to stakeholders in business applications and integrated solutions, giving us tests of functionality from end to end. Often, scenarios are essential for business acceptance, because they encapsulate test ideas in a format that is meaningful for business users and easy for them to understand and review.

User stories, use cases, and other business requirements can be good sources of scenario test ideas. But testers know that these are rarely comprehensive or detailed enough to encompass a thorough test without additional analysis. And if we base our test model entirely on the same sources used by the programmers, our test will reflect the assumptions they made building the system. There is a risk that we will miss bugs that arise from misinterpreted or incomplete requirements or user stories.

One way to mitigate this risk is to build a scenario model whose foundation is a conceptual framework based on the data flows. We can then build scenarios by doing structured analysis of the data. This method helps to ensure adequate coverage and testing rigor, and it provides a cross-check for our other test ideas. Because it employs a structure, it also facilitates building scenarios up from reusable components.

Definitions

Scenario

One dictionary defines “scenario” as:

> An outline or model of an expected or supposed sequence of events.[1]

In his Introduction to Scenario Testing, Cem Kaner extends his definition to testing:

> A scenario is a hypothetical story, used to help a person think through a complex problem or system…A scenario test is a test based on a scenario.[2]

Test Model

Every software test is based on a model of some kind, primarily because we can never test everything. We always make choices about what to include in a test and what to leave out. Like a model of anything—an airplane, a housing development—a test model is a simplified reduction of the system or solution we are testing.

The test model we employ embodies our strategic choices, and then serves as a conceptual construct within which we make tactical choices. This is true whether or not

[1] The American Heritage® Dictionary of the English Language: Fourth Edition, 2000.

[2] Cem Kaner, An Introduction to Scenario Testing, 2003.
http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf

we are aware that we are employing models. Every test approach is a model—even if we take the “don’t think about test design, just bang out test cases to match the use cases” approach. But if we’re not consciously modeling, we’re probably not doing it very well.

Here’s Jerry Weinberg’s general definition of ”model”:

> Every model is ultimately the expression of one thing…we hope to understand in terms of another that we do understand.[3]

In these definitions of “scenario” and “model”, we see the inextricable connection between scenarios and models. For our purposes, then:

> A **test model** is any reduction, mapping, or symbolic representation of a system, or integrated group of systems, for the purpose of defining or structuring a test approach.
>
> A **scenario** is one kind of **model**.

We don’t have to use scenarios to model a software test, but we do have to model in order to use scenarios.

Designing a Model for a Scenario Test

It is useful to model at two levels when we are testing with scenarios:

> The **overall execution model** for testing the solution.
>
> The **scenarios** that encapsulate the tests.

There are many different ways to model both levels. The retail Point of Sale (POS) system example in this paper uses a hybrid model based on **business operations** combined with **system data** as the basis for the overall test execution model, and **system data** for the scenario model.

- **Business operations**, e.g.
  - “Day (week, month, mock-year) in the life”
  - Model office or business

  A business operations model is the easiest to communicate to all the stakeholders, who will immediately understand what we are trying to achieve and why. Handily, it is also the most obvious to construct. But it is not always the most rigorous, and we may miss testing something important if we do not also look at other kinds or levels of models.

- **System Data,** categorized within each flow by how it is used in the solution and by how frequently we intend to change it during testing:

[3] Gerald M. Weinberg, An Introduction to General Systems Thinking, p.28

- Static
- Semi-static
- Dynamic

Combining different model bases in one test helps testers avoid the kinds of unconscious constraints or biases that can arise when we look at a system in only one way. We could also have based our models on such foundations as:

- The **entity lifecycle** of entities important to the system, e.g.:
  - product lifecycle (or account, in a banking system)
  - customer experience

- **Personae**[4] defined by testers for various people who have some interaction with the system, or who depend on it for data to make business decisions. These could include people such as:
  - merchandising clerk
  - store manager
  - sales associate
  - customer
  - category manager

- High-level **functional decomposition** of the system or organization, to ensure that we have included all major areas, including those that may not have changed but could be impacted by changes elsewhere:
  - Functional areas within the system or business (Ordering, Inventory Management, Billing, etc.)
  - Processes in each area (Order capture, provisioning, etc.)
  - Functions within each process (Enter, edit, cancel order, etc.)

- Already defined **user stories** or **use cases**

- **Stories** created by testers, using their imaginations and business domain knowledge, including stories based on particular metaphors, such as:
  - soap opera tests[5]

Although none of these was central in our test model, we used each to some degree in our modeling (except for pre-defined user stories and use cases, which didn't exist for this system). Using a range of modeling techniques stimulated our thinking and helped us avoid becoming blinkered by our test model.

Modeling Based on Data

We constructed our primary central model using a conceptual framework based on the system data. This is a good fit for modeling the scenario test of a transactional system.

[4] See books by Alan Cooper for an explanation of persona design.
[5] Hans Buwalda, *Soap Opera Testing*; Better Software, February 2004.

A data-driven model focuses on the data and data interfaces:

- Inputs, outputs and reference data
- Points of entry to the system or integrated solution
- Frequency of changes to data
- Who or what initiates or changes data (actors)
- Data variations (including actors)

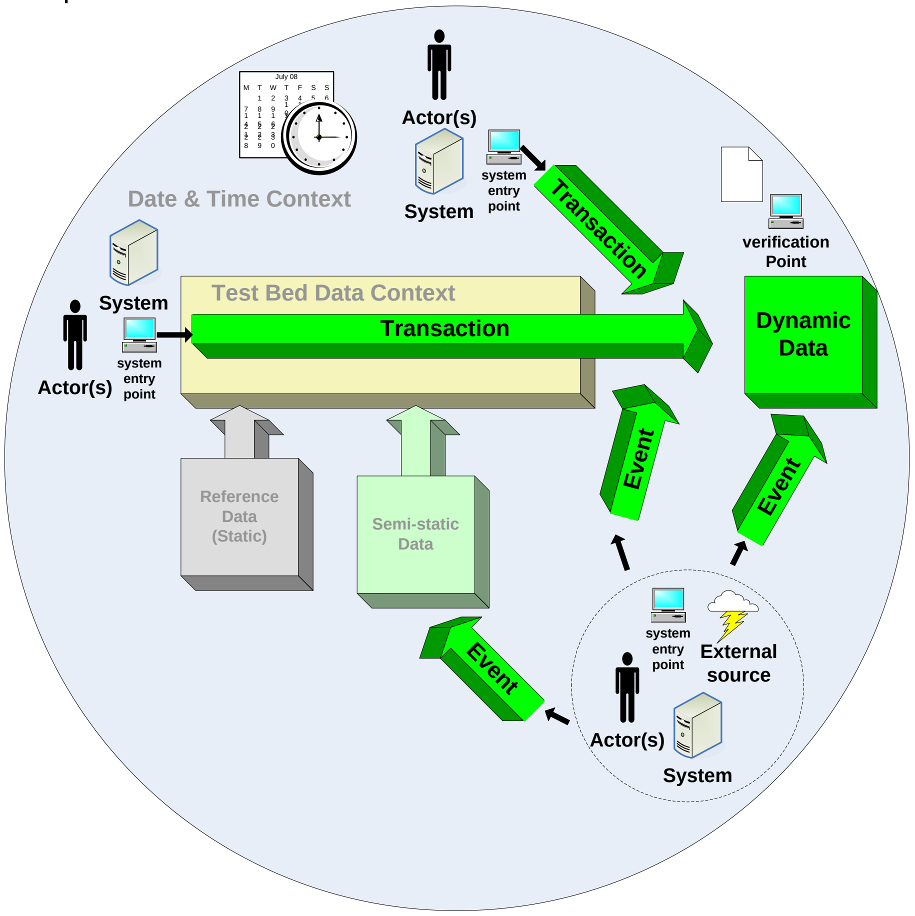
A model based on data represents a systems view that can then be rounded out using models based on business views. As well as providing a base for scenarios, focusing on the data helps us identify the principal components we will need for the overall execution model for the test:

- Setup data
- Entry points
- Verification points
- Events to schedule

It's also easy to structure and analyze.

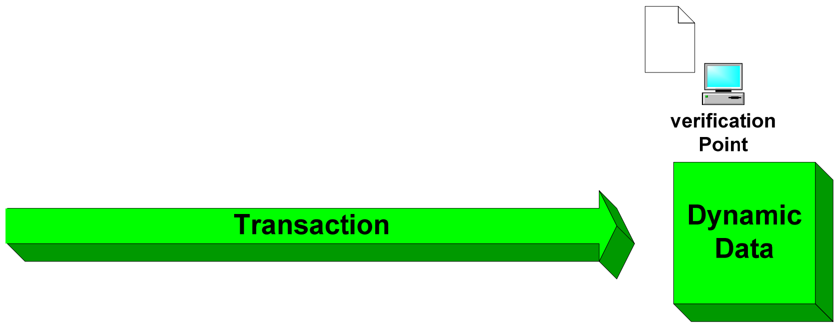
Conceptual Framework for a Data-driven Model

Here's a conceptual framework..



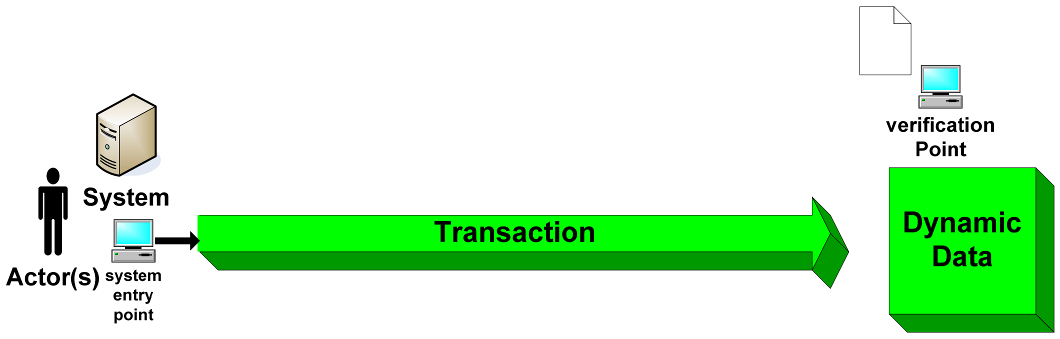
At the most basic level, we want scenarios to test the outcome of system transactions.

Transactions drive dynamic data, i.e., data that we expect to change in the course of system operation. Transactions represent the principal business functions the system is designed to support. Some examples for a POS system are sales, returns, frequent shopper point redemptions, customer service adjustments.



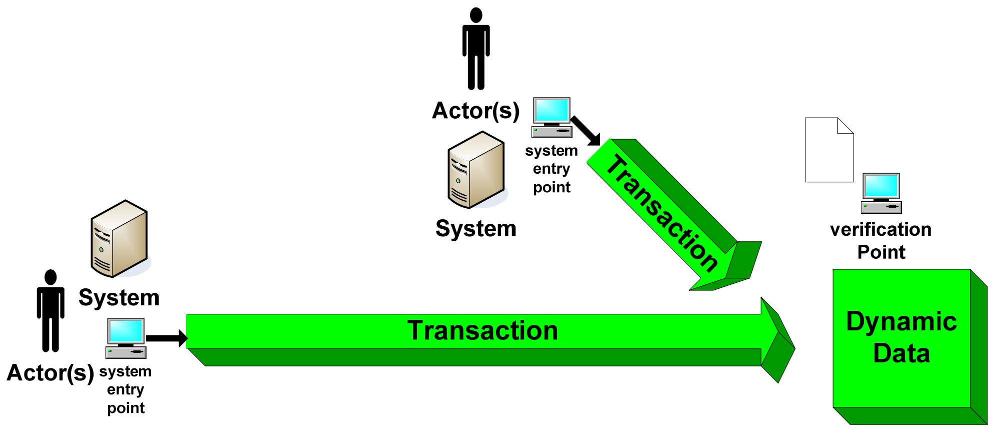
A transaction is initiated by an **actor**, which could be human or the system.

There may be more than one actor involved in a transaction, e.g., a sales associate and a customer:



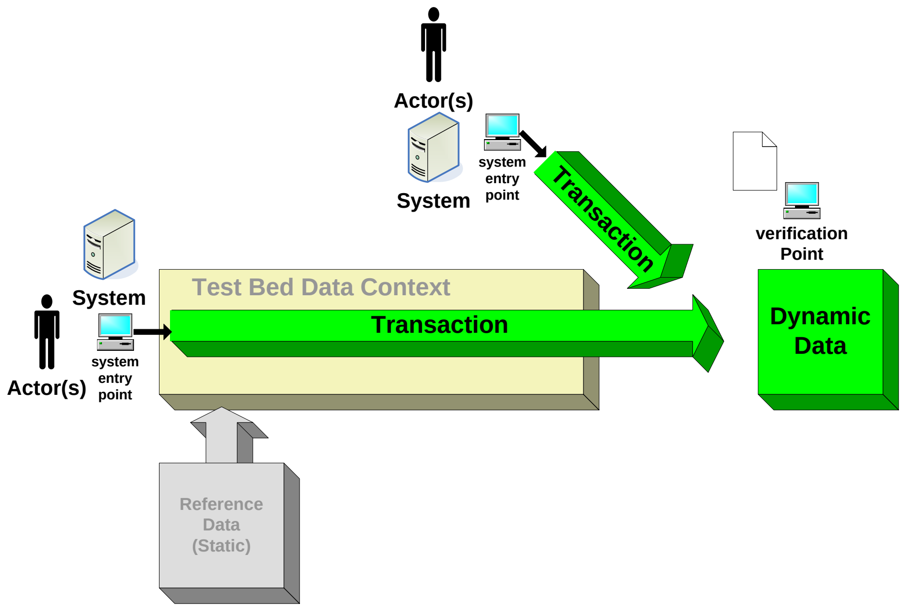
Many scenarios will have **multiple transactions**.

Subsequent transactions can affect the outcome by acting on the dynamic data created by earlier related
transactions.



Transactions operate in a context partly determined by **test bed data**.

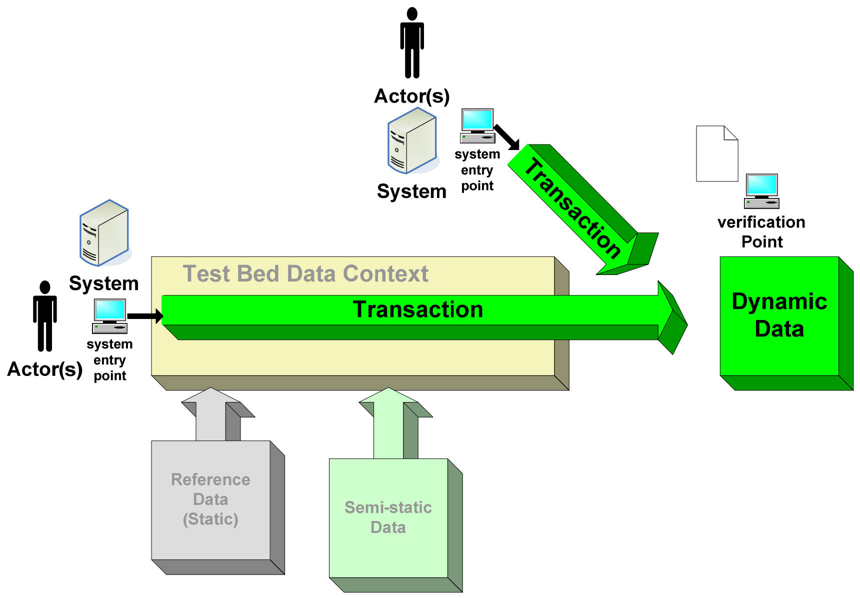
Reference test bed data is static (doesn't change in the course of the test), e.g., user privilege profiles, frequent shopper (loyalty) points awarded per dollar spent, sales tax rates. Deciding which data should be static is a test strategy decision.



The test bed also contains **semi-static data**, which can change the context and affect the outcomes of transactions.

Semi-static data changes occasionally during testing, as the result of an event. Examples of semi-static data include: items currently for sale, prices, and promotions.

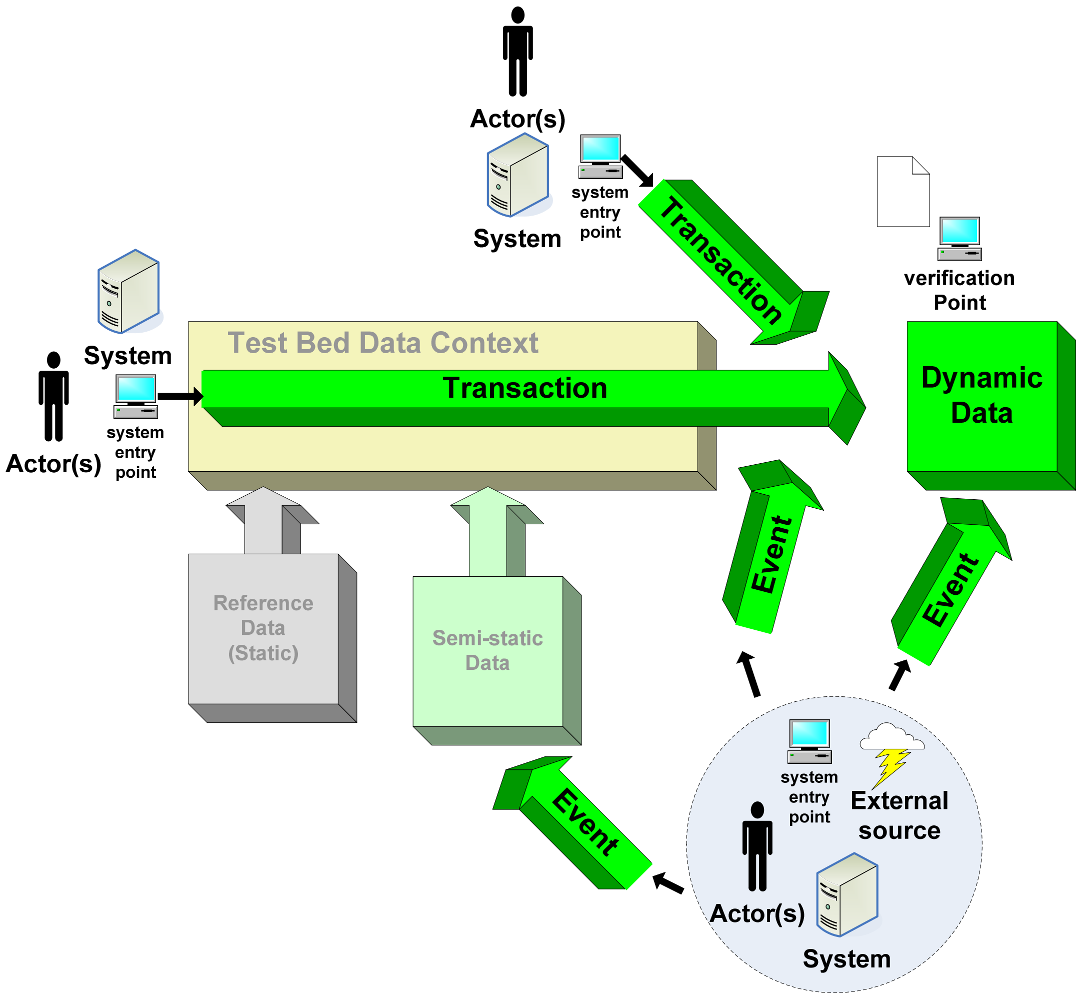
Determining which data will be semi-static, and how frequently it will change, is also a test strategy decision**.**



Events affect transaction outcomes—by changing the system or test bed data context, or by acting on the dynamic data.

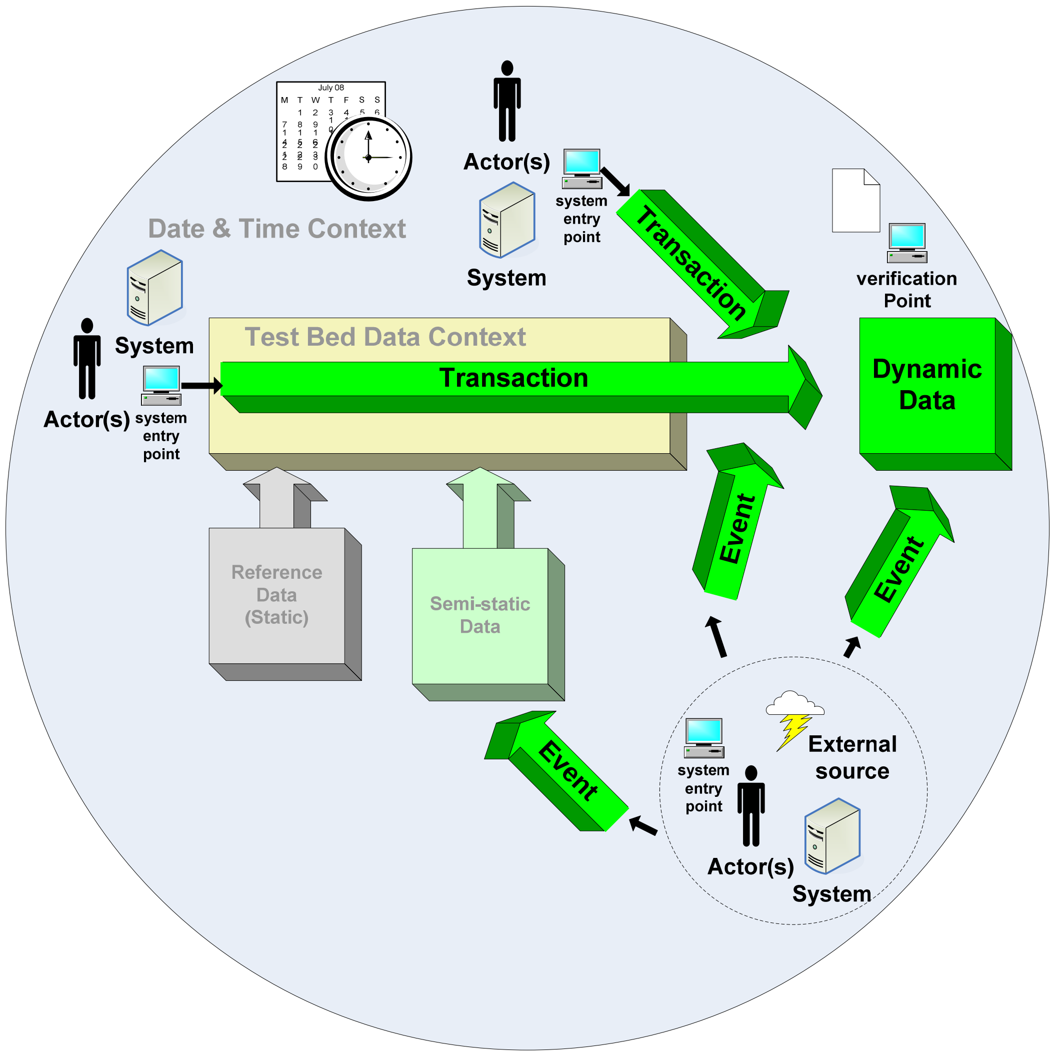
Events can represent:

- periodic or occasional **business processes**, e.g., rate changes, price changes, weekly promotions (deals), month-end aggregations
- **system happenings**, such as system interface failures
- “external” **business exceptions**, such as a shipment arriving damaged, or a truck getting lost



Scenarios also operate within a context of **date and time**.

The date/time a transaction or event occurs can have a significant impact on a scenario outcome.



Categorizing the data determines each type's role in the test, and gives us a conceptual framework for the scenario model:

- Reference data sets the context for scenarios and their component transactions.
- A scenario begins with an event or a transaction.
- Transactions have expected results.
- Events operate on transactions and affect their results
    - A prior event changes a transaction context, e.g., an overnight price change.

    - A following event changes the scenario result and potentially affects a transaction, e.g., product not found in the warehouse.

- Actors influence expected results, e.g., through:
    - User privileges
    - Customer discount or tax status

We can apply this framework to the design of both levels of test model: the overall execution model (the central idea or metaphor for the test approach), and the scenario model.

Test Design for a Point of Sale (POS) System

The example that follows shows how, together with a small test team, I applied the conceptual framework described above to testing a POS system on a client project. A core member of my team was David Wright, a senior tester with whom I have worked on many systems integration test projects, and who has contributed many ideas and practical details to development of this method. As well as experience with end-to-end scenario testing, David and I both have extensive retail system domain knowledge, which was essential to successful testing on this project.

Background

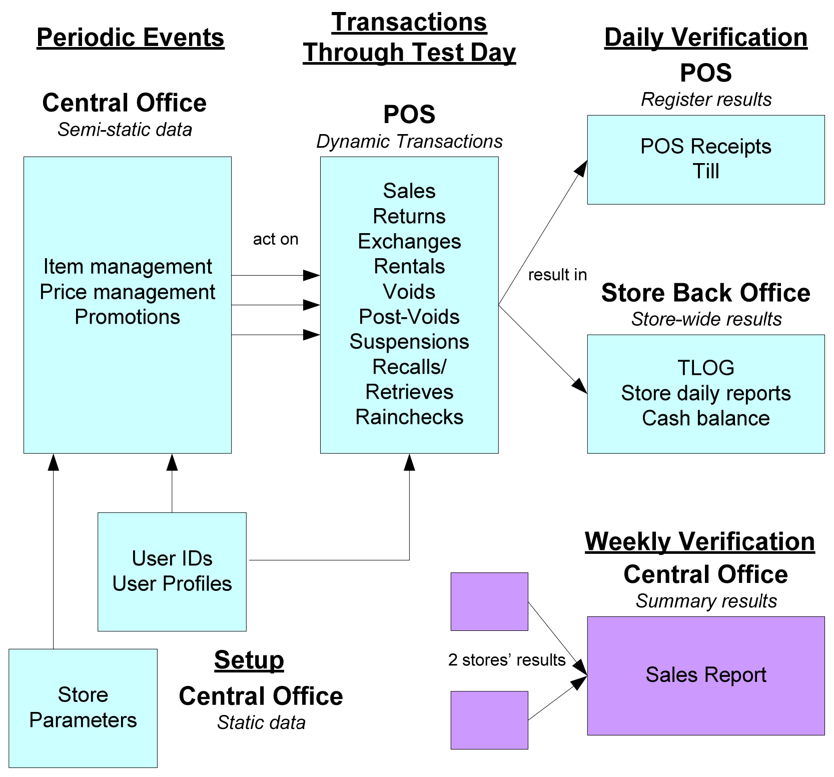
The client operates a chain of more than 1000 retail drugstores across Canada. This project was for their secondary line of business, a home health care company (HHC), which sells and rents health care aids and equipment to retail customers through 51 stores in different regions of Canada. The POS system was being implemented standalone for the HHC. Following successful implementation for HHC, the plan was to implement POS in the main drugstores, fully integrated with an extensive suite of store and corporate systems.

The POS system is a standard product, in production with several major retailers internationally, but it was being heavily customized by the vendor for this client. The client had contracted with an integrator to manage the implementation and mitigate its risks. I reported to the integrator with my test team.

The vendor was contractually obligated to do system testing. The integrator had a strictly limited testing budget, and there were several other constraints on my test team’s ability to develop sufficient detailed knowledge of POS to perform an in-depth system test within the project’s timelines. I therefore decided that our strategy should be to perform an end-to-end acceptance test, focusing on financial integrity (i.e., consistency and accuracy end-to-end). Because we expected the client would eventually proceed with the follow-on major integration project in the drugstores, I built the test strategy and the artifact structure to apply to both projects, although an estimated 40% of the detailed scenario design would need to be redone for the primary line of business.

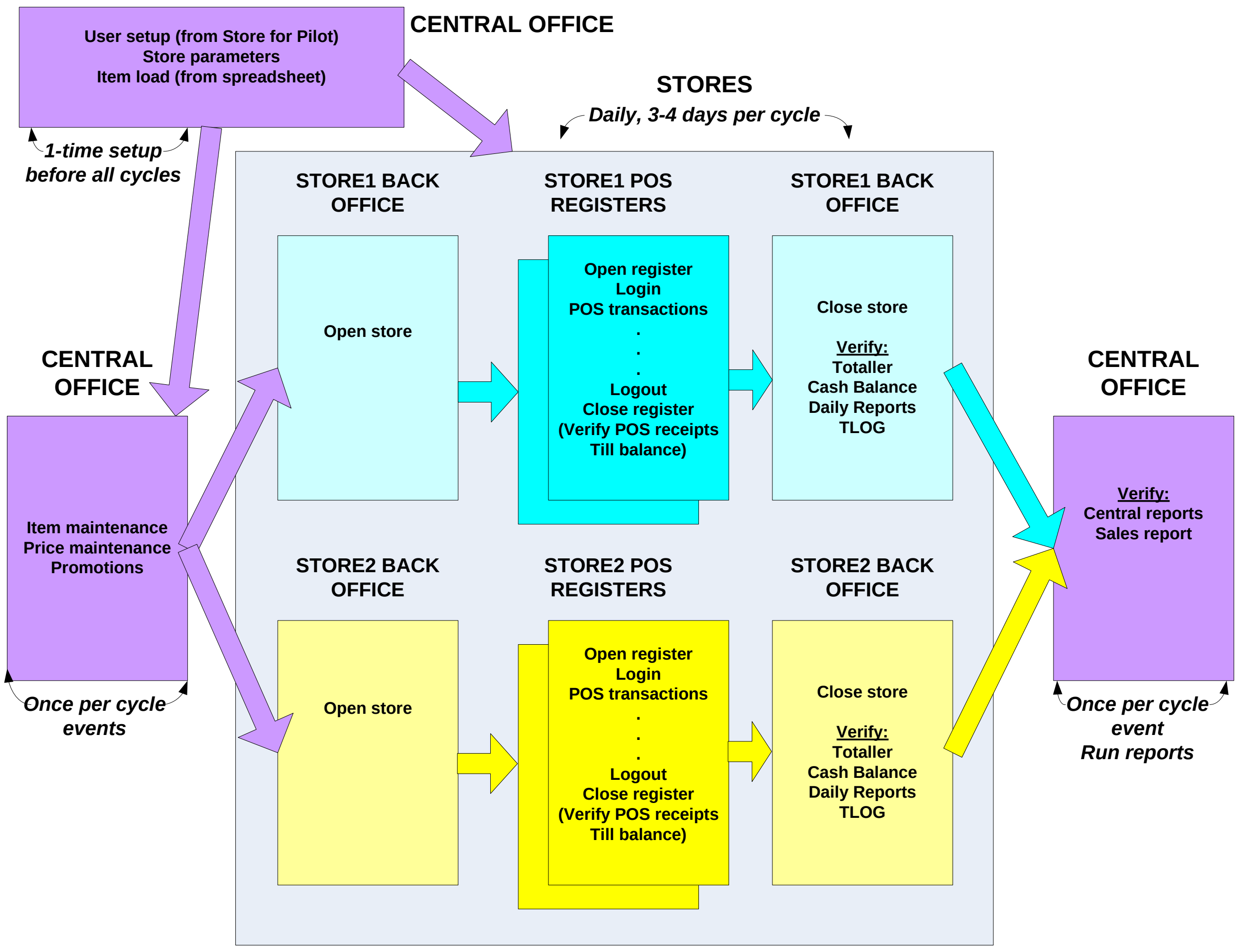
Overall Execution Model

Having analyzed and categorized the data for the POS system, I designed this overall model for executing the test.



The POS system had four principal modules (Item and Price Management, Promotions Management, Store Operations, and Central Reporting), operating in three location types as shown in the diagram. The POS client would operate in each store and Store Back Office. The Central Office modules would be the same for all stores and all would offer the same items, but actual prices and promotions would differ by region and be fed overnight to each store. We decided to run 2 test stores, covering 2 regions with different pricing and provincial tax structures. One of our test stores could be switched to a different region if we identified problems requiring more in-depth testing of location differences.

We combined a business operations view with our data-driven overall model. This gave us a test cycle model that looked like this.



Scenario Design

We decided to build our scenarios up from an element we defined as an item transaction. The team began by drilling down on each framework element and defining it from a test point of view: i.e., important attributes for testing, and the possible variations for each. Item transactions had some other elements listed (actors, items) with them, giving us a list that looked like this:

Transaction

- Type (sale, return, post-void, rental, rental-purchase…)
- Timeframe (sale/rental + 3 days…)
- Completion status (completed, void, suspended)
- Store
- Register
- User (cashier, manager, supervisor, store super-user)
- Customer (walk-in, preferred, loyalty, status native, employee…)
- Item(s) [multiple attributes, each with its own variations]
- Tender (cash, check, credit card, debit, coupon, loyalty redemption…)

- Loyalty points (y/n)
- Delivery/pickup (cash and carry, home delivery…)
- Promotions in effect (weekly flyer, store clearance…)
- Other discount (damage…)

Using spreadsheets to structure our analysis, we designed a full range of item transactions, working through the variations and combining them to make core test cases. Wherever possible, we built in shortcuts, e.g., Excel lists for variations of most attributes.

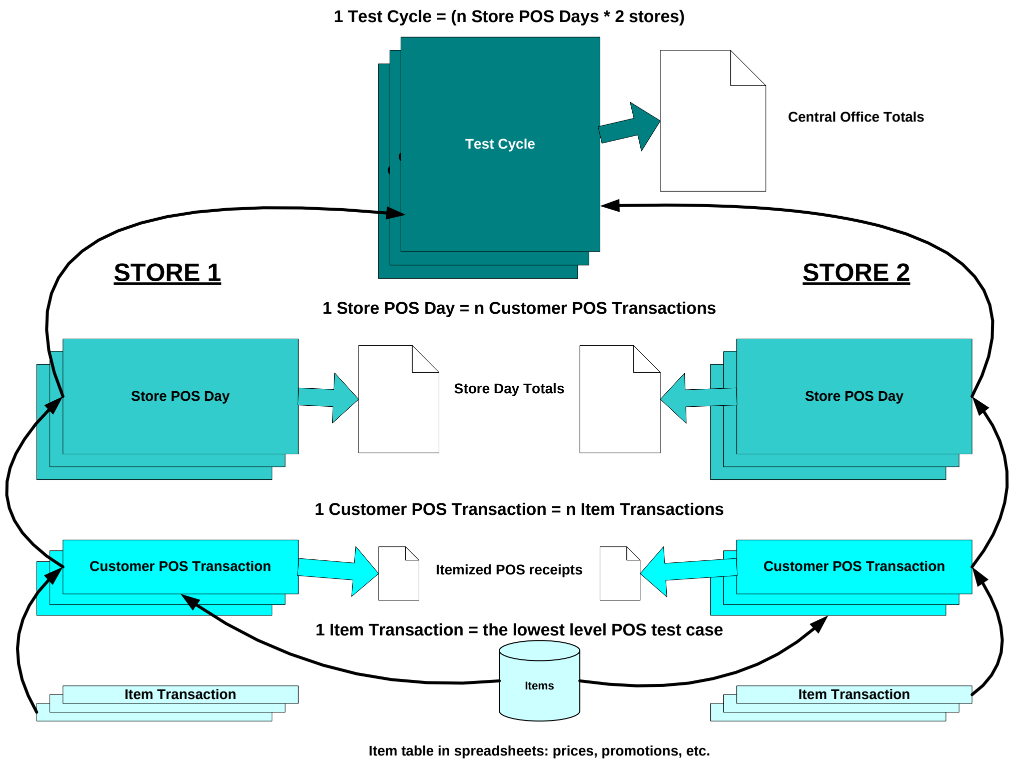
Here's a simplified example of the transaction definitions.

Txn-ID	Store	Type	Timeframe	Completion	Register	User Profile	Customer	Item(s)	Tender	Points	Delivery /Pickup	Promos	Discounts
POS-1	1	sale	n/a	C	4	cashier	walk-in	45270	cash-CAD	n	C&C	n/a	n/a
POS-2	1	return	sale+1	C	2	manager	walk-in	45270	cash-CAD		n/a	n/a	n/a
POS-3	2	sale	n/a	c	1	cashier	employee	98651 54945 21498	Visa	y	HD	sidewalk	raincheck

We designed events in a similar fashion, including price changes and promotions. We also constructed an item table with all their attributes and variations.

We then designed scenarios, using the item transactions we'd defined as the lowest-level building blocks. This allowed us to combine item transactions and also to use them at different points in scenarios.

Here's the scenario model:


1 Test Cycle = (n Store POS Days * 2 stores)
Test Cycle
Central Office Totals
STORE 1
STORE 2
1 Store POS Day = n Customer POS Transactions
Store POS Day
Store Day Totals
Store POS Day
1 Customer POS Transaction = n Item Transactions
Customer POS Transaction
Itemized POS receipts
Customer POS Transaction
1 Item Transaction = the lowest level POS test case
Items
Item Transaction
Item Transaction
Item table in spreadsheets: prices, promotions, etc.

The scenario spreadsheets referenced the item transaction and item sheets, so we could make changes easily. Our spreadsheet structures also allowed us to calculate expected results for each test day and test cycle. This was essential for a system with so many date dependencies, where we had to evaluate the cumulative results of each store sales day daily, and at the end of each cycle evaluate the corporate week for all stores.

Here's a simplified version of the scenario design

Scenario ID	Description	Prior txns	Prior events			Main Txn	Follow-on Txns	Post-Txn Events
			Central Office	Store Back Office	Store			
POS-S-S1	Senior phones in to buy 3 items on Super Senior's Day of which 1 is a charge sale item and 1 is a govt funded item, delivery and service charges applied, pre-paid delivery	n/a	PROMO-45 PROMO-7	Open Store	Open Register Login	POS-S133		Logout Close Register Close Store RPT-16
POS-S-S19	Loyalty customer pays for 5 items, of which 4 are points-eligible; customer changes mind before leaving register and 1 points-eligible and 1 non-eligible are post-voided	n/a	n/a	Open Store	Open Register Login	POS-S17	POS-PV17	
POS-S-S65	Senior phones in to buy 3 items on Super Senior's Day of which 1 is a charge sale item and 1 is a govt funded item, delivery and service charges applied, pre-paid delivery	n/a	PROMO-12 PROMO-7 ITEM-96	Open Store	Open Register Login	POS-S133	n/a	`

In addition, we wrote Excel macros to automate the creation of some test artifacts from others. The testers' worksheets for test execution, for example, were automatically generated. (Unfortunately, the worksheets are too large to show an example legibly in this format.)

Results of the POS Test

The vendor POS system, and in particular the customizations, turned out to be very poor quality. As a result, a test that had been planned to fit our budget, with 4 cycles (including a regression test) over 4 weeks, actually ran through 28 cycles over 14 weeks—and then continued after pilot implementation. My strategy, and our test scenarios, proved effective. We logged 478 bugs, all but 20 of which our client considered important enough to insisted on having fixed:

Severity	Count	%
1	8	**2%**
2	331	68%
3	116	25%
4	23	5%
Total	**478**	

It's important to ask whether this was the most cost-effective way to find those bugs. Probably it was not, since many of them should have been found and fixed by the vendor before we got the system. But many other bugs would probably not have been found by any other kind of test. And—given the total picture of the project—it was critical that the integrator conduct an independent integrated test. This was the most efficient way for our team to do that with the constraints we had.

There were several business benefits from our test and test strategy. The most important was buy-in from the client's Finance department. Because our test verified financial integrity across the system, it provided Finance with evidence of a solid solution from their point of view.

Our scenario test facilitated acceptance for implementation in the home health care stores, and provided information for the decision on whether or not to proceed with integration and implementation in the 1000+ drugstores.

Because we tested the end-to-end function of complex data interactions that are business norms for this client, such as sales, returns and rentals of items with date-dependent prices and promotions, and returns when prices have changed and new promotions are in effect, we were able to provide the business with an end-to-end view of system function that crossed modules and business departments. Our varied scenarios provided sufficiently realistic data to verify critical business reports.

Finally, the spreadsheet artifacts we used throughout the test and gave to the client supplied solid evidence of testing performed, in the event it should ever be required for audits.

Conclusion

Testing Benefits on the POS Project

The principal benefit of my strategy from a testing point of view was that my team supplemented the vendor's testing rather than attempting to duplicate it. The vendor's testing did not include an integrated view of POS function. Ours did, and that allowed us to focus primarily on system outcomes, and only secondarily on the users' immediate experiences.

By adopting a building-block approach, we incorporated efficiency in test preparation and execution. This gave us flexibility to reschedule our testing according to the state of the system on any given day. When we encountered bugs, we could work with the vendor and drill down to the components of a scenario (item setup, item transaction, promotion setup, etc.) to find the problem.

Our robust and structured transaction-scenario artifacts provided the client with a reusable regression test for future releases, upgrades to infrastructure, etc.

We were able to layer operability tests on top of our scenario testing, simulating real conditions and verifying the outcomes.

When to Consider Scenario Testing

Scenario testing as I have described it here is not always the best test method. It does not fit well with exploratory testing, and it is not the most effective way to test deep function. It is, however, a very useful method for testing widely across a system. It is better for testing system outcomes than for evaluating a user's immediate experience.

Scenario testing should, therefore, be considered as part of an overall test strategy that includes different kinds of tests. Some situations where it could be appropriate include:

- Acceptance tests of business systems, e.g., UAT or vendor acceptance.
- End-to-end systems integration tests of multi-system solutions, or of enterprise integrated systems.
- Situations where a test team lacks sufficient detailed system knowledge to do under-the-covers or deep function testing and has no way to get it, and insufficient time to explore. When there is a combination of inadequate documentation, restricted or zero access to people who wrote the software, and critical time pressures, scenario testing might be the best solution to the testing problem. (All of these constraints applied on the POS project, and we overcame them with scenario tests informed by business domain knowledge.)

Critical Success Factors

The single most important requirement for designing good scenario tests is business domain knowledge, in the form of one or both of:

- Testers with experience in the domain
- Input from, and reviews of scenarios by, business representatives

Where neither of these is available, it is at least possible to resort to industry books, as Cem Kaner's suggests[6].

To use the approach described in this paper, you need:

- A model with a framework that fits the type of application. A data-driven model works well for transactional systems. For other types of applications, e.g., a desktop publishing system, you would need to create a different model and framework, such as one based on usage patterns.
- Testers skilled in structured analysis. This is not the no-brainer it sounds. Not all testers—not even all good testers—have this skill.
- A building-block approach, so you can design and build varied and complex tests from simple elements. Among other advantages, this allows you to begin testing with the simplest conditions before adding complexity to your scenarios. It also makes it possible to automate some of the test artifacts, and build in calculated expected results. This becomes essential in large-scale systems integration tests, where you have to communicate accurate expected results to multiple teams downstream in the integration.

[6] Cem Kaner, An Introduction to Scenario Testing, 2003.
http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf

Risks

If you adopt scenario testing, these are the critical things to watch out for.

- Scenario testing can miss important bugs you could find with under-the-covers or deep function testing. Remember that scenario testing is better for testing end-to-end than it is for testing deeply, and build your strategy accordingly.
- Bugs found in scenario outcomes can be difficult to diagnose. Especially if system quality is suspect, it is essential to begin with simple scenarios that test basic end-to-end function, only proceeding to complex scenarios when you have established that fundamental quality is present.
- In choosing a model, there is always a risk of fixing on one that is too restrictive. Applying two or more model types will help prevent this.
- Equally, there is a risk of choosing a model that is too expansive (and expensive).

Finally, never fall in love with one model type. This applies to your models for a single test as much as it applies to your model choices for different tests. Every test is different, and every model type can bring benefits that others lack.

.