

Handling Technical Debt with OutSystems

TECHNICAL WHITE PAPER



Contents

Introduction	
OutSystems Versus Technical Debt	4
OutSystems From Development to Deployment	
TrueChange™	
Compare and Merge	7
Architecture Dashboard	
The Architecture Canvas	
Architecture Canvas Validation	
Using the Architecture Dashboard	
Conclusion	
Appendix: Architecture Design for Architecture Dashboard	
Validation Rules	
No Upward References	
No Side References Between End Users	
No circular references	
Application Composition Rules	
No Upward References in Applications	
Clarify Ownership	
Avoid Mixing Sponsors	

Introduction

Since Ward Cunningham coined the phrase "technical debt" in 1992, it has sometimes been taken as an excuse for releasing quick-and-dirty code in the interests of development speed. Instead, it is better understood as an explanation for why it is sometimes OK to get a workable solution out the door—with a commitment to find an optimal one as soon as possible, and, in this way, repay a technical debt.

However it is defined, taking on technical debt plays an important part in achieving excellence—as long as it is redeemed by bringing everything up to spec before it leads to a cascade of future problems. Like the financial world, from which this metaphor is drawn, ignoring long-term debt ultimately leads to overburdened, bankrupt enterprise software.

Even the best code has room for improvement—but, before enhancements can be tackled for the next release, disruptive technology may mature, markets may pivot, and additional user needs may be identified. Paying down a large backlog of technical debt at the beginning of a development cycle may block the quick response required to deal with new opportunities and challenges.

OutSystems Versus Technical Debt

Applications built with OutSystems rely on standard architectures and frameworks. They don't require proprietary components, runtime engines, or interpreters—so technical debt is effectively limited before development even begins.

Software deployment using classical programming methods typically comprises modifying database files, pre-processing code, compiling it, assembling it, and then linking the result to all of its function calls. Bugs in any step may only be revealed as the result of a failed compilation, which triggers a manual code review for troubleshooting.

OutSystems orchestrates the entire deployment process using a combination of automation, AI, and analytics to identify architecture errors, faulty logic, and broken dependencies—during development, in real time. OutSystems preempts failed compilations by either applying corrective measures automatically or directing the developer to the source of the error—along with a suggestion about how to fix it. Modules can only be published when it will function correctly.

Correct functioning, however, does not necessarily mean efficient functioning. While a short-term solution may solve the immediate problem, it may also leave a technical debt, in the form of an app that is difficult to maintain and costly to change. The OutSystems Architecture Dashboard provides a high-level analysis of your app, marks areas for improvement, and facilitates unhindered growth.

This paper discusses how OutSystems helps reduce technical debt during development and assists in building a best-practices architecture to efficiently deal with it in the future.

OutSystems From Development to Deployment

OutSystems promotes and encourages the use of Agile methodologies, so it is extremely important to allow developers to be highly productive and efficient not only when creating applications but also when changing them to adapt to new business needs.

Service Studio, the Outsystems visual development environment, gives developers the tools they need to achieve these goals.

TrueChange™

When a developer makes changes to a model, the TrueChange[™] engine provides immediate feedback on its impact. By inspecting the model for consistency—a method similar to syntactic and semantic analysis used in classical coding—TrueChange checks if the model is well-formed and well-typed, and, at the same time, detects many other problems, including those that may lead to performance issues, marking them accordingly for followup. In the image below TrueChange has identified two errors, noted three unused arguments and given a warning about a security concern. Clicking links in the TrueChange tab brings the developer to the issue to be corrected. **1-Click Publish** remains disabled until all errors have been resolved.



Developers may also **debug applications** by setting **breakpoints** at specific points in a module and then running the logic step-by-step to find any issues in logic design.



The **Debugger** tab shows app information, including variable and runtime values, along with current debugging context (current thread, event name, UI flow, screen and action). The developer can use the **Debugger Toolbar** and the menu to track down and correct the problems.

Compare and Merge

OutSystems uses a versioning system that labels modules with an incremental version number upon each successful publish. In environments where many developers work on the same module at the same time, **1-Click Publish** first initiates a comparison of the new code with the existing code before going forward.

<u>Compare and Merge</u> first tries to merge new code with the changes that other developers published in the meantime. If automatic integration is not possible, the developer can review the results in the **Compare and Merge** window, as seen in the image below. Drilling down into conflicts, shown in red, displays the changes between the versions. Once the conflicts have been reconciled and all of the merges approved, the app can then be published.

Compare and Merge	□ ×
Select the elements to keep from either version.	Submit Feedback
Double click all element to see details	
Your version (uploaded at 1:55 PM)	The other version (uploaded at 2:09 PM by ashley)
MergeDemo	1 MergeDemo
✓ → UI Flows	UI Flows
MainFlow	MainFlow
Style Sheet (merged with conflicts)	CSS Style Sheet
Section	Section
Preparation	Preparation
Section (merged with conflicts)	Section
CSS Style Sheet Modified by admin at 2:05 PM	CSS Style Sheet Modified by ashley at 1:58 PM
Advanced	Advanced
Style Sheet span.intro {	Style Sheet span.intro {
Last modified by admin at 2:05 PM	Last modified by ashley at 1:58 PM
MERGE AND PUBLISH	MERGE HELP Cancel

Architecture Dashboard

Enterprises have moved away from legacy monolith systems—where a single stack contains all of the functionality of an application—so they can more quickly answer customer needs, respond to changing operational requirements, and meet business goals and objectives.

Breaking the monolith means breaking out functionality and services so they can be modified independently. This allows for the frequent release of new features and the ability to roll out updates as soon as vulnerabilities are discovered.

But this is only one part of the task. These services must continue to work together, and some of the pieces broken off from the original monolith may require ever greater effort as time goes on to keep the app running, turning into mini-monoliths themselves. This is technical debt par excellence: in down-toearth developer lingo, spaghetti architecture.

Among the main issues of this technical debt include the following:

- **Poor service abstraction:** Services for core business concepts are not correctly isolated and abstracted. Business rules are spread over different systems, making them hard to reuse, and code reuse has very little structure.
- Unmanageable dependencies: Components of the system are not correctly isolated from each other, so updating or replacing a system has a snowball effect on other systems.
- Inflexible, slow-moving legacy components: Legacy systems can be complex and inflexible, and may include obsolete technology. Adapting such systems to business changes is slow and difficult. And the accumulation of core information and system dependencies over time can hinder any replacement.



A well-considered application architecture provides patterns and techniques for design and development: a roadmap and best practices for building a wellstructured app. Beyond that, application architecture helps drive consensus among all players, supports planning, facilitates change, and manages complexity, all of which reduces risks and successfully manages technical debt.

It is fundamental to align architecture with business goals: to translate business needs and concepts into an architecture that allows developers to evolve and scale up with business, while ensuring independent life cycles between different lines of business.

While developing a conceptual design for application architecture is beyond the scope of the article, information about this subject can be found in the links below:

- Designing the Architecture of Your OutSystems Applications
- Web and Mobile Architecture with Architecture Dashboard (presentation)
- Integration Patterns for Core Services Abstraction
- Domains and Services Architecture

The Architecture Canvas

The architecture canvas, as used by the OutSystems **Architecture Dashboard**, is a multi-layer framework that provides a systematic approach to speed up architecture design. It promotes the correct abstraction of reusable services and components, maximizes independent life cycles among all the parts of your architecture by setting the correct mutual dependencies, and minimizes the impact of changes, making it much easier to maintain and evolve your apps.

End-User	No Services	User Interface and Processes Provide functionality to end-users
Core	Reusable Services	Business Services Services around business concepts
Foundation	Reusable Services	Non-Functional Requirements Services to connect to external systems or to extend your framework

All modules are sorted into layers as follows:

- Foundation layer
 - Business agnostic, reusable, non-functional modules
 - Connections to external systems, such as SAP and external databases
 - Framework extensions, such as libraries, reusable UI patterns and themes
 - Non functional requirements, such as auditing

• Core layer

- Systems agnostic (modules on the foundation layer abstract integration details)
- Reusable services around business concepts
- Business entities, business rules, business transactions, business widgets
- Recommendations for architects:
 - No front-end screens
 - Read-only entities (write via APIs)

- End-user layer (to support user interaction)
 - User interfaces and processes, reusing core and library services to implement the user journey
 - Independent end-user applications (to ensure lifecycle independence)
 - Recommendations for architects:
 - No business logic
 - No core entities

The figure below illustrates how an OutSystems app with a well-formed architecture integrates with a typical enterprise IT ecosystem.



Architecture Canvas Validation

Architecture Dashboard is the OutSystems technical debt monitoring tool. Powered by **AIFusion™** and **CodeDNA**, it automatically performs code and runtime analysis before recommending solutions for improving the performance, security, architecture, and user experience of applications.

A guided refactoring capability powered by the CodeDNA engine identifies all of the opportunities for refactoring and code duplication across the factory, pointing architects and teams to the right points to make these improvements. Code duplication is one of the top technical debt issues. The guided refactoring capability is unique in our industry. It uses AI to detect, with incredible accuracy, where teams should focus on refactoring and ranks them by importance.

Architecture Dashboard uses the following validation rules in order to evaluate the architecture of your applications:

- No upward references
- No side references among end users
- No circular references between the foundation and core layers



For an in-depth discussion of these rules for architecture design see the Architecture design **appendix** below.

Using the Architecture Dashboard

The Architecture Dashboard is designed for:

• Architects who need to understand how the architecture evolves

• **Team leads** who need to make sure that there are no problems with the architecture

• **Developers** who need to make sure their code adheres to best practices and doesn't introduce any performance or security issues

Architecture Dashboard provides an integrated, bird's eye view of technical debt across an entire portfolio of applications and the interdependencies between modules in the developers' environment. The OutSystems AI engine classifies every architecture module into its correct class, automating architecture discovery. A heat map visualization of mild-to-severe problem areas helps IT leaders identify problem areas quickly and to prioritize them accordingly.

outsystems • Architec	ture Dashboar	d									DevOps Training	• @ ■	TA TK A	
	<													
Q Search for app		Apps												
eams		End user												
All teams	•	Apps Common Web Traditional Web	IdP Traditional Web	rield Services Resources Traditional Web	Floor Plan Traditional Web	rield Services Backoffice Reactive Web	Order Management Multi-language Reactive Web	Demo Setup Traditional Web	SupplierPortal Traditional Web	Smart Power Backoffice Traditional Web	Order Approval Mobile	Salesforce Connector Traditional Web		
Open apps report		ZXing Services Traditional Web	Drag And Drop Mobile Mobile	Google Places Reactive Reactive Web	Input Mask Reactive Reactive Web	JsBarcode Mobile Reactive Web	Smart Power Portal Reactive Reactive Web	Ultimate PDF Traditional Web	ExecDashboard Traditional Web	Directory Traditional Web	Demo Team Template Screens Mobile	MyCustomApp Mobile		
		psoTestMobileApp Mobile	Mobile Banking Mobile	MyOtherCustomA Mobile	Twilio Connector Traditional Web	Signature Traditional Web	Field Services Telco Traditional Web	Credit Backoffice Traditional Web	MyChatbot Traditional Web	WeatherApp Traditional Web	MyApp Traditional Web	Okta SAML Test Traditional Web	i –	
		Fleet Management Traditional Web	Involce Management Traditional Web	My Test Traditional Web	ReactiveOrderMa Reactive Web	Smart Power Mobile								
		Core												
		Field Services Mobile Mobile	Smart Power Core Service	Employee Services Service	Apps Common Mobile Mobile	Smart Power Resources Traditional Web	Geo Traditional Web	Firebase Cloud Message Traditional Web	Profile Utils Traditional Web	Field Services Alexa Connector Traditional Web	Social Collaboration Core Traditional Web	Chatbot Web Traditional Web		
		CRM Services Service	BAM Reports Reactive Reactive Web	Order Management Traditional Web	Properties Services Traditional Web	Twilio Video Reactive Reactive Web	OutSystems AI Language Analysis Traditional Web	Twilio Video Utilities Mobile	Twilio Video Mobile Mobile	TouchTracker Mobile	Lottie Animations Reactive Web	Google Analytics web Traditional Web		
		Architecture Dashboard Traditional Web												
		Foundation											_	
		Azure Cognitive Services Connector Mobile	Firebase Cloud Message Plugin Mobile	Motion and Orientation Plugin Mobile	Calendar Plugin Mobile	Card IO Plugin Mobile	AuthLib Service	OutSystems UI Web Style Guide Traditional Web	Speech Recognition Pluain Mobile	Image Resizer Plugin Mobile	CameraPreview Plugin Mobile	Google Analytics Reactive Reactive Web		
		Azure Bot Framework	Azure LUIS Connector	App with external DB connection	Architecture Dashboard Shared	ardoHTTP	Data Grid	EnvironmentSetti	FileSystem	Firebase Core	GraphViz Services	HashUtils		
		May 9, 2021	Sinal Web	Library	Traditional Web	Library	Library	Service	Library	Library	Library	Library	- +	
		18.	73%	Randomizer Number Generator Library	SAP Utilities Library	Simple QR Code Generator Library	Time Zone Traditional Web	TrigFunctions						100
		Last 2 week	s											Ē
											Latest sync (on May 9, 2021	10:07:45	A

Clicking an app shows its dependencies and technical debt with other applications.

<								
Q. Search for app	Apps							
ams	End user	_						
All teams 👻	Apps Common Web Traditional Web	ndø Traditional Web				Smart Power Backoffice Traditional Web		
pps Common Web ×	ZXing Services	Drag And Drop Mobile						
aditional Web app o team								
gility: 🛑 🚯								
Open app report								
Open modules (3)	Core							
			Apps Common Mobile Mobile	Smart Power Resources Traditional Web				
	Foundation							
	May 9, 2021	S prad Web						- +
	and the second s	anal Plugin						

Double-clicking one of the modules allows you to drill down into any identified issues, which are grouped by **Architecture**, **Performance**, **Maintainability** and **Security**, so you can quickly understand what is going on in that particular area.

outsystems • Architectu	ure Dashbo	ard						DevOps Training 👻 🕐 📕 🐼 TK AD
Back to Apps								
Report								Export to Exce
pps		Modules		Category				
Apps Common Web	*	SampleApp_UsersCore	*	All categories			*	Duplicated Code Duplicated logic in different action flows
nalyzed		Findings status		User				
Any time	*	All status	*	All users			*	By repeating the same logic in different action flows you increase the cost of maintaining your code. For example, to
								modify this logic later, you'll need to find and modify the code in all the different action flows.
ode patterns					1=			How to Fix
Public Entities aren't read-	only			Architecture	7		î	Perform an assessment for all the findings inside pattern, including that you have access to all of their occurrences. (i not, please talk to your Team Lead or Factory Architect)
End-User application provi	ding service	25		Architecture	1		L	Patterns
Duplicated				Maintainability	2	>		Q Pattern #5: 3 duplicates Opened •
Unlimited records in SQL q	uery			Performance	8		L	Q Pattern #13: 2 duplicates Opened
Dynamic inline parameter				Performance	3		L	
o j i i i i i i i i i i i i i i i i i i								
Unlimited records in Aggre	egate			Performance	3		L	
Exposed REST services are	not secured	1		Security	1		L	
Missing description on pub	olic element			Maintainability	28			
Large resource				Performance	1		~	
								Latest sync on May 9, 2021, 10:07:45 AM /L

Architecture Dashboard explains the impact of this finding and gives a recommendation about how to fix it. In the screen below clicking the duplicated code pattern shows three instances of where duplicated code is being used, along with links to Service Center where this code can be found.

Back to Apps			
eport	Pattern #5: 3 duplicates		× Export to Exc
ps pps Common Web	🖸 🏪 🔴 SetPictureUserApp 🛛 >		^
alyzed	🖸 🍓 🔴 AddPictureByGuidToUser	Null espace?	
ode patterns	🖆 🍖 AddOrChangePictureToUser		code. For example, to
Public Entities aren't rea		GetAppidByEspaceId GetAppIdByEntryEspaceId	of their occurrences.
End-User application pro			
Duplicated Code			Opened
Unlimited records in SQ		PictureUserApp PictureUserApp ApplicationId ApplicationId	opened
Oynamic inline paramet		Ġ Ġ	
Jnlimited records in Age			
Exposed REST services a		Set PictureUserApp	
Aissing description on put	olic element	Maintainability 28	

Conclusion

OutSystems applications rely only on standard architectures and frameworks and require no proprietary components, runtime engines, or interpreters. For these reasons technical debt is effectively limited before development even begins.

When developers write new code, or make changes to existing code, OutSystems provides immediate feedback on its impact. Corrective measures are applied automatically if possible. Otherwise the source of the error is linked, along with suggestions about how to fix it. This eliminates compilation errors, since an OutSystems app cannot be published unless they function correctly.

Correct functioning, however, does not necessarily mean efficient functioning. While a short-term solution may solve the immediate problem, it may also leave a technical debt, in the form of an app that is difficult to maintain and costly to change. The OutSystems Architecture Dashboard provides a high level analysis of your app, highlighting areas for improvement, and facilitates unhindered growth.

Appendix: Architecture Design for Architecture Dashboard

This appendix contains a detailed discussion of rules for canvas validation and application composition.

Validation Rules

The following validation rules are used for correct architecture design of all of the modules in your development environment.

No Upward References

An upward reference tends to create a cluster where any two modules, directly or indirectly, have a circular dependency.



In the example above, foundation module X—by definition non-functional should not depend on functional or end-user modules.

Another unexpected effect of this upward reference from foundation module X is that end-user 2 (EU2), by consuming core module B, makes it dependent on EU1's cluster. Aside from giving ER2's runtime an unnecessarily large footprint, it will also be impacted by changes made in modules which have no relevance to its function.

No Side References Between End Users

End-user modules should not provide reusable services, as seen in the figure below. The side reference between EU1 and EU2 means that their lifecycles are linked and they can only be released at the same time.



Maintaining correct isolation between end-user modules give them independent lifecycles with a versioning pace set by their different sponsors or project teams.

No Circular References

A cycle is always undesirable, since it brings unexpected impacts and hard-tomanage code. A cycle between modules indicates that the concepts are not correctly abstracted.



Application Composition Rules

A business application is generally understood as a computer program used to perform one or more business functions. In the context of architecture design in OutSystems, however, an application is considered a set of modules that constitutes a minimal deployment unit.

So, in addition to an end-user application, which fits the common definition for a business app, developers also deploy foundation applications, composed of a set of modules other apps can use for foundation services, and, in a like manner, core apps to supply core services.

The type of application is defined by the modules in the topmost row, as can be seen in the figure below:



Just as modules must be designed using the validation rules, so composing applications from these well-formed modules must be layered in the same way.



- No upward references
- No side references between end users
- No circular references

No Upward References in Applications

The example below shows two end-user applications. Both the first and second project are well formed. However, the third project, which combines both of them together, has an unacceptable upward reference in its design.



Because end-user applications should not provide services to other applications, this validation error is corrected by creating a new core application that isolates the common service to which Application 1 and Application 2 connect, as seen in the figure below:



Clarify Ownership

Having more than one owner for an application with several modules results in complex deployment management, as accountability for what has been changed becomes unclear. The top portion of the figure below illustrates the difficulty in determining who is responsible for what.

After splitting the common application into two apps ownership is now clearly defined.



Avoid Mixing Sponsors

For the same reason, avoid mixing sponsors. As an example, consider a portal that allows executing simulations over different insurance lines of business. If all LOBs are under the same application, none of them can be released independently, meaning that the slowest LOB dictates the release cycle.

The right side of the figure shows that creating independent applications per line of business—and a portal application with a fourth sponsor—allows each LOB to determine its own pace of delivery.

