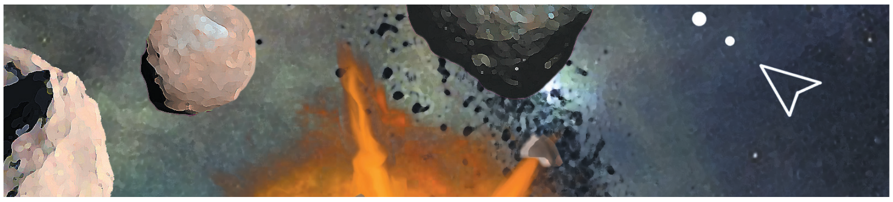
Improving Digital Game Development with Software Product Lines

Andre W.B. Furtado, Andre L.M. Santos, and Geber L. Ramalho, Federal University of Pernambuco

Eduardo Santana de Almeida, Federal University of Bahia

// A systematic process for exploiting software product lines for game development offers both domain-specific languages and generators streamlined for game subdomains. //



SOFTWARE REUSE and family-based production strategies, including software product lines (SPLs),[1] make software engineering more effective and productive. Such strategies let software designers and engineers analyze and implement systems collectively rather than separately, automating more of the software life cycle through reusable domain assets such as application blocks, frameworks, patterns, domain-specific languages (DSLs), generators, and tools.

Domains ranging from consumer electronics to avionics have successfully applied such concepts (see www.splc.net/fame.html). However, several factors have hindered digital game developers from doing the same:

- the engineering is the hardest part of game development;[2]
- the field is characterized by ad hoc, low-level development;[3]
- game developers struggle with integrating components and managing their architectural complexity;[4] and
- game development isn't the same as software development, so traditional requirements engineering isn't applicable, and the popular concept of "game genres" can be misleading in an SPL process because they're ambiguous and imprecise.[5]

Yet expectations for digital games are extremely high,[4] and game development involves many technical and design risks. Each new wave of games implements unproven technical features without knowing how players will react. Game developers and designers thus should focus on such risks and on features that make the game unique; they shouldn't waste their time repeatedly performing menial and routine tasks.

To enable game developers and designers to successfully apply SPL processes to game development and overcome the aforementioned challenges, we describe a practical SPL-based approach for analyzing digital game domains and implementing core domain assets—an area that other SPL and domain-engineering processes don't comprehensively address. Our approach should help game developers create DSLs and generators, key SPL components still underexplored in the context of game development.

Beyond Game Engines

Game engines are a state-of-the-art resource for game development. They sup-

0740-7459/11/$26.00 © 2011 IEEE

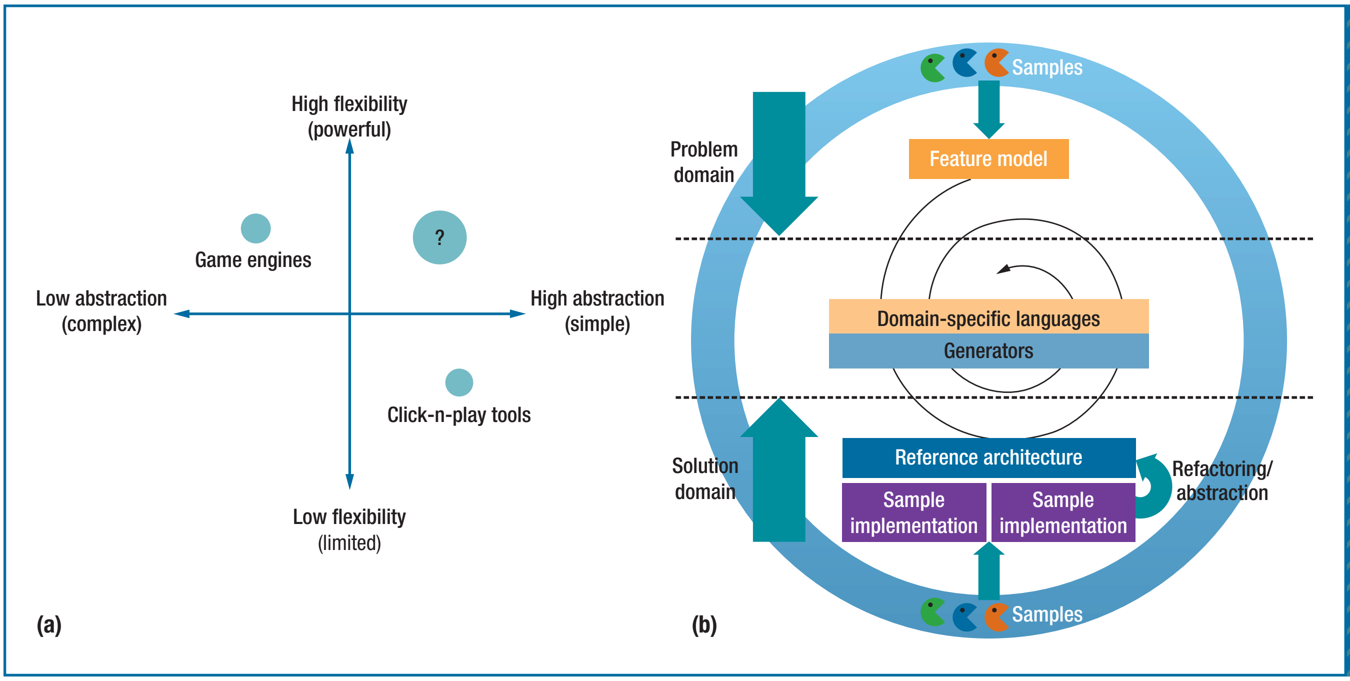


FIGURE 1. Automating game development. (a) Game development's current hiatus, caused by a lack of simple yet powerful tools, and (b) a domain-specific approach that creates core domain assets for a game software product line (SPL).

port common foundations such as texture rendering, world management, and event handling. However, understanding a game engine's architecture, interaction paradigm, and programming peculiarities usually isn't simple or intuitive. Furthermore, the development environments used with game engines might not provide all the desired development foundations for specific game domains.

Yet game engines could play a more important role in automating game development. Consider Don Roberts and Ralph Johnson's recurring pattern on automating software development:[6]

1. identify reusable abstractions for a domain and document patterns for using abstractions;
2. develop runtimes and frameworks to codify abstractions and patterns; and
3. define languages and build tools to support the runtime and frameworks, such as (visual) editors and compilers.

Game engines are situated in the second phase of this "pattern-runtime-language" workflow. Although a framework such as a game engine can reduce the cost of developing an application by an order of magnitude, mapping the requirements of each product variant onto the framework is a nontrivial problem, generally requiring the expertise of an architect or senior developer.[6] Language-based tools (phase three) automate this mapping step by capturing variations in requirements via language expressions and encapsulating the abstractions a framework defines. This helps users think in terms of the abstractions and generates framework completion code. Such tools also promote agility by expressing domain concepts (such as a digital game's screens and entities) in a way that customers and users better understand and by propagating changes to implementations more quickly.

Rather than simply disappearing, multimedia APIs such as DirectX and OpenGL became the foundation upon which game developers built more abstract layers, such as game engines themselves. Similarly, game engines could become the foundation for more abstract and expressive layers, such as DSLs and generators, integrated with the development environment. This could help end the current hiatus in game development, caused by the fact that easy-to-use script languages and click-n-play tools, such as Game Maker and RPG Maker, aren't flexible enough, while game engines are powerful but complex (see Figure 1a).

Domain-Specific Game Development

SPL processes often target software development in general, but they can be more effective when bound by macrodomains. So, we conceived a domain-specific approach for game development that uses game domain analysis to create core assets—such as DSLs and reference architectures—for a game SPL.

TABLE 1 Example of nonemotional game features traceability.

Problem domain features	Solution domain features
Allow breaks to avoid having players lose progress	Save/load, pause/resume, "continue"
Register player performance	High-scores table, achievements
Provide social interaction	Multiplayer mode (online and local)
Establish a player identity	Avatars, game elements customization
Provide availability (to play independent of time/space)	Mobile platforms, digital convergence (multidevice experience for the same game)
Ensure readiness to play	Intuitive/one-click installers, zero-deployment games
Offer replay value	Multiple narrative paths, multiplayer support, achievements
Establish a low learning curve	Tutorials, scaffolding (hints and tips that stop being offered as players acquire experience)
Advertise a specific brand (typical for advergames)	Hooks for brand insertion, which can end up as patterns, such as the background of "loading" screens, midaction fly-outs, specific areas or canvas designated for branding, and so on
Teach or train the player on a given real-world topic	Missions and problem-solving challenges that incorporate the topic contents, notorious in serious games and educative games

The proposal builds on strict top-down and bottom-up domain-engineering approaches, combining them in a spiral and iterative *edge-center* process. In other words, the process avoids the risk of a big upfront investment in any edge (problem or solution domain) by having game developers and designers work through such domains in iterations, incrementally (see Figure 1b).

Avoiding a strict top-down approach lets game developers and designers identify abstractions from the product code, so the generated code isn't just a convenient, direct output from DSL constructs, like a serialization. Avoiding a strict bottom-up approach ensures DSLs aren't solely based on code template expressions, architectural abstractions, and refactorings. It also improves abstraction by enabling the DSLs' syntaxes and semantics to be more than just a (graphical) representation of the code.

The proposed approach also considers the peculiarities of digital games. For example, it employs game engines as a vital piece in defining a reference architecture. Moreover, the approach helps developers and designers explore numerous game samples, which are widely available thanks to the diversity of platforms, an intensive prototyping culture, an abundance of user-generated content, and player nostalgia.

Here, we outline our approach in a linear form, but practitioners should perform the activities in iterations—analyzing samples, extracting features, inspecting code, and modeling or implementing assets in increments.

Envision the Game Domain

The first task of planning a digital game SPL should be to envision the domain. Instead of relying solely on the concept of game genres, the game SPL designers should describe expectations for predefined *core game dimensions* that aren't overly specific or generic: *player* (number of players, co-playing modes, score system, and so on), *graphics*, *flow* (levels, screens, rooms, and so on), *entities*, *events*, *input*, *audio*, *physics*, *AI*, *networking*, and any custom dimension for the target game domain, such as the *battle system* for a role-playing game (RPG) domain.

Analyze the Game Domain

Domain analysis is the process of identifying, capturing, and organizing the information used in developing software systems and making that information reusable for new systems.[7] Here, we offer some guidelines for game domain analysis.

Identify emotion-based requirements. When modeling the problem domain, identify the emotion-based requirements (such as immersion, surprise, and nostalgia) and handle them through experimentation and prototyping. However, keep in mind that nonemotional requirements can still exist for games (see Table 1).

Select domain samples. Taking into account the expectations defined for the core game dimensions and nonemotional requirements, select domain samples by identifying existing, under development, or anticipated games that belong to the target domain. Select the most representative games, such as those re-released through "remakes," those that have numerous sequels, or those with broad industry and media recognition.

Define and refine game domain features. Analyze the samples and evolve the core game dimensions to domain-specific features. We recommend feature modeling to express the commonality and variability extracted from samples.[8] When analyzing game sequels (such as Pac-Man and Ms. Pac-Man), it's possible to consider unique sequel features as extensions or variations of the original game.

Explore "locked" game features. To explore locked game features (functionalities hidden until players make enough progress), consider enabling "god modes," activating "cheat codes," exploring official and "underground" literature related to the game (such as the World of Warcraft wiki), and interviewing experienced players.

Create subdomains. Consider partitioning the target game domain into subdomains (for example, partitioning a broader *arcade* domain into *shooter* and *maze* subdomains). The individual analyses of more specific subdomains lead to more expressive and effective SPL assets, such as DSLs.

Anticipate future features. Game domain experts and analysts can extend the feature model with innovative features to enhance the game SPL. Recommended feature anticipation techniques include retrospection, trend analysis, and the morphologic box,[9] whose unusual and unforeseen combinations of values cope with creativity—an essential component of game design.

Assess the Automation Potential

By grouping analyzed features into cohesive sets, game SPL analysts can identify subdomains to assess their automation potential. Analysts should break down identified subdomains into even more fine-grained, atomic subdomains—for example, the transition between game screens, the collision relationship between game entities, or graphical representations of heads-up displays.

When identifying game subdomains, practitioners should

- consider core game dimensions and the features directly derived and elicited from them as subdomain candidates;
- rely on the knowledge of domain experts to further break down the game samples' characteristics;
- investigate how types (such as classes, interfaces, and enumerations) are modularized in sample implementations and game engines—modules and submodules can provide hints on possible subdomain candidates; and
- investigate repetition in sample implementations,[10] such as design patterns or a piece of design or code that repeatedly appears in a sample or across samples, even if the repetition instances aren't identical.

Finally, game SPL designers should prioritize subdomain candidates for automation, considering their

- previous automation evidence (modeling languages and tools, such as generators) and how easy it is to integrate them into the game SPL;
- coverage (the amount of features a given subdomain covers);
- productivity (developer effort saved by automating the subdomain); and
- abstraction (how much less error-prone the subdomain will be after automation).

We don't believe in a one-size-fits-all game architecture. Reference game architectures must be built for their target (sub)domains.

In contrast with the approach's high-level game domain analysis, which occurs in the problem domain, some of these guidelines require lower-level technical tasks, which happen in the solution domain: investigating code samples to extract patterns and templates, implementing prototypes for anticipated features, and creating a reference architecture. But such low-level tasks shouldn't be delayed until after domain analysis, as in waterfall processes. Both edges (the problem and solution domains) meet halfway, leading to more effective core assets.

Create Application Assets

Application core assets (such as components, frameworks, and prototypical applications) are building blocks for SPL products. In the context of game SPLs, we suggest building a domain-specific game architecture from the composition and adaptation of the game engine (or engines) created or already used for the target game domain. As opposed to the architecture Eelke Folmer proposed (see the "Related Work in Game Development" sidebar),[4] we don't believe in a one-size-fits-all game architecture. In other words, reference game architectures must be built for their target (sub)domains—but not from scratch.

When building the reference game architecture, practitioners should promote it to a *domain framework*,[11] which is a reusable SPL component

RELATED WORK IN GAME DEVELOPMENT

We can find some level of domain-specific development in game engines because they evolved from APIs into a more comprehensive toolset encompassing script languages, such as UnrealScript. However, such languages are still at a fairly low programming level, raising concerns as to the level of abstraction they offer.[1]

Eelke Folmer applied component-based development to game development, establishing a commercial off-the-shelf approach in which a layered reference architecture is suggested for all digital games, with limited reuse areas such as graphics and sound.[2] However, his process is strictly bottom-up—it doesn't consider the problem domain and related tasks such as domain analysis.

Jeroen Dobbe introduced a new domain-specific language for computer games, hosted in its own special environment and integrated with a proprietary game engine.[1] His experience provides lessons learned on applying DSLs to digital games but doesn't define a process or guidelines for performing domain-specific game development.

Emanuel Reyno and José Cubel proposed using model-driven development through platform-independent models and platform-specific models to create prototype 2D platform games for PCs.[3] They acknowledge that UML diagrams are more familiar to software engineers than game developers.

Frank Hernandez and Francisco Ortega developed a DSL instance for modeling 2D games.[4] Their work diverges from ours on two fronts: they believe the 2D gaming domain is specific enough to be expressed by a single DSL and that game engines should be consumed as is by the generated code (instead of adaptation layers).

Sonja Maier and Daniel Volk[5] discuss first findings of a case study in which language workbench concepts are applied to create level editors for "classic" games, such as Pac-Man. The authors aren't concerned with defining a process for creating game factories; rather, they focus on a specific subdomain (level editing) in which only one DSL is created per "factory."

Finally, Leandro Nascimento[6] defined a practical approach for implementing core assets in a mobile game software product line.

References

1. J. Dobbe, "A Domain-Specific Language for Computer Games," MSc dissertation, Dept. of Software Technology, Delft Univ. of Technology, 2007.
2. E. Folmer, "Component Based Game Development: A Solution to Escalating Costs and Expanding Deadlines?" *Proc. 10th Int'l ACM SIGSOFT Symp. Component-Based Software Eng.*, Springer, 2007, pp. 66–73.
3. E.M. Reyno and G.A.C. Cubel, "Model-Driven Game Development: 2D Platform Game Prototyping," *Proc. Game-On 2008, 9th Int'l Conf. Intelligent Games and Simulation*, EUROSIS, 2008, pp. 5–7.
4. F.E. Hernandez and R.R. Ortega, "Eberos GML2D: A Graphical Domain-Specific Language for Modeling 2D Video Games," *Proc. 10th SPLASH Workshop on Domain-Specific Modeling*, Aalto-Print, 2010; www.dsmforum.org/events/DSM10/papers.html.
5. S. Maier and D. Volk, "Facilitating Language-Oriented Game Development by the Help of Language Workbenches," *Proc. 2008 Conf. Future Play: Research, Play, Share*, ACM Press, 2008, pp. 224–227.
6. L.M. Nascimento, "Core Assets Development in Software Product Lines: Towards a Practical Approach for the Mobile Game Domain," MSc dissertation, Center of Informatics, Federal Univ. of Pernambuco, 2008.

that encapsulates mandatory (common) subdomain features identified during domain analysis. The domain framework is consumed by artifacts generated from diagrams (DSLs) used to model the variable subdomain features. Promoting a game engine and its encompassing reference architecture into an SPL asset, however, might not be a straightforward task, unfolding into different possibilities: reusing the game engine as is, implementing one from scratch, or creating an adapter layer.

Ultimately, a game engine promoted to a domain framework should support three important requirements. The first one is the target game domain's variability space, so that a game engine predicts variation points and effectively supports their implementation. The second one is framework completion—that is, the game engine exposes an interface that's expressive and concise enough so that code that consumes (configures) it can be easily generated via model-driven development (MDD) techniques. Finally, the promoted game engine should be extensible, so that developers can complement the game SPL's built-in feature set with custom code.

Create Development Assets

Development core assets, such as DSLs, are integrated by SPL processes into a highly customized development environment to provide guidance, automation, and abstraction to the product development. To create DSLs in the context of game SPLs, game SPL designers should characterize the variability of the identified game subdomains, previously prioritized for automation. The variability spectrum ranges from routine configuration (that is, configuring a product using simpler, tree-like DSLs, such as wizards or feature-based configuration, to select a subset of features) to creative construction (which involves using textual or visual languages to create complex, graph-like DSLs, such as programs and models).[12]

The characterized variability helps game SPL designers to create the subdomain DSL's abstract and concrete syntaxes. Then they can define trans-

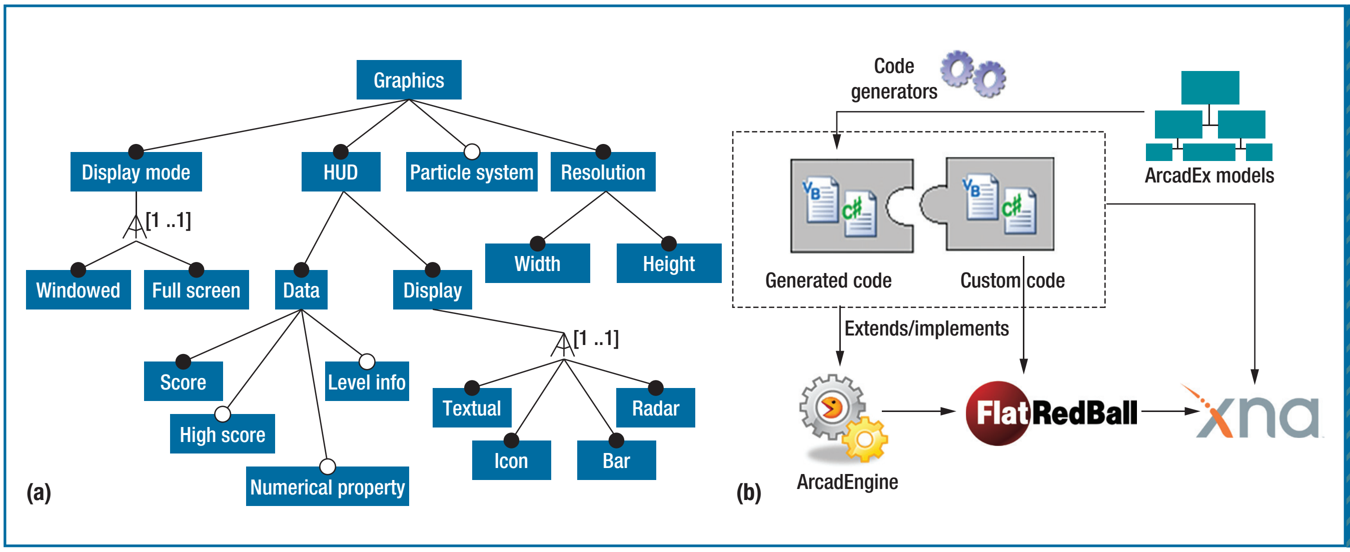


FIGURE 2. The ArcadEx implementation. (a) A feature model subset for the ArcadEx game SPL (the "graphics" concept) and (b) ArcadEx core assets.

formations (generators) by investigating mappings from elements in the domain framework (game engine) code to elements in the DSLs. During this inspection, new information might need to be included in the original DSLs before they can be used as inputs to the transformations (for example, a game screen might introduce details such as the main character's start position). Once the DSL is refined, the game SPL designer produces a template-based code generator by migrating the code from the reference implementation to templates, annotating it with tags and scriptlets that bind the code to the DSL.

Finally, game SPL designers should design and plan the development environment integration, including the project and build system, custom property editors, semantic validators (to catch modeling errors), and contextualized automated guidance (suggestions and guidance on game development activities to be performed in a particular context).

An SPL for 2D Arcade Games

We used our proposed approach to create game SPLs targeted at different game macrodomains, such as isometric adventure games, RPG games, mobile touch-based games, and 2D arcade games. Here, we discuss ArcadEx, an SPL we created for 2D arcade games.

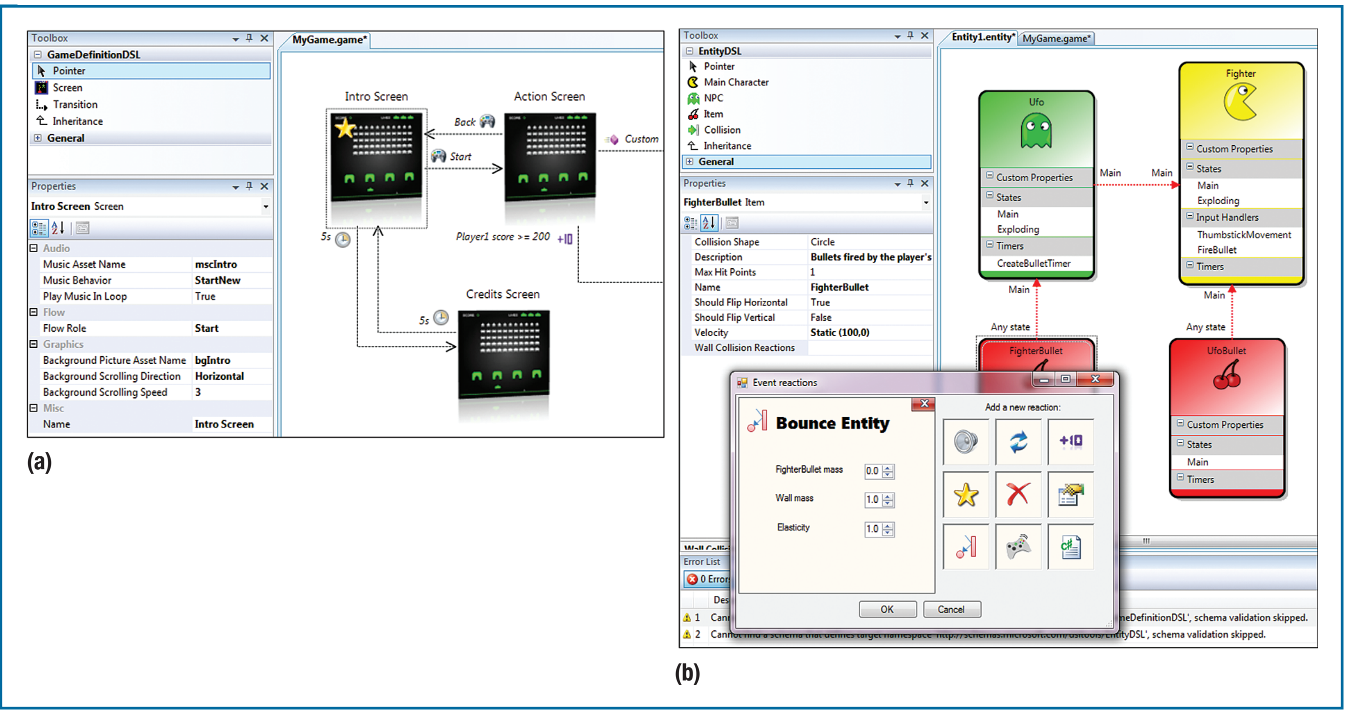
Implementation

The core game dimensions defined for ArcadEx describes it as a game SPL focused on single or multiplayer 2D arcade games, with short levels composed by screens containing entities and walls. By using keyboard or gamepad-based controllers, players control main characters who collide with other entities such as nonplayer characters and items.

The ArcadEx implementation followed the proposed edge-center iterations. In the high level, each iteration included a round of game domain analysis, focused on detailing the feature model of one or a couple of prioritized game subdomains at a time. For instance, Figure 2a shows the feature model of the "graphics" concept, resulting from the first iterations. Eventually, we built a feature model with almost 150 features to describe the domain's commonality and variability. We analyzed approximately 30 games, with each analysis taking, on average, two to four hours. Our domain-analysis guidelines were especially useful for discarding samples and filtering out conflicting features.

In the lower level, we investigated the most representative samples of each iteration subdomain from an implementation perspective. As Figure 2b shows, we used the FlatRedBall game engine, which consumes the XNA framework, to implement samples. We gradually promoted the engine to a domain framework, implementing and expanding an adaptation layer, ArcadEngine, to cover each iteration subdomain. ArcadEngine not only complements FlatRedBall with specific features of the target ArcadEx game domain but also enables the game engine to be more seamlessly consumed by code generated from models. In other words, ArcadEngine moves complexity away from the code generators.

Finalizing each iteration, the high- and low-level work met halfway, culminating with the design and implementation of one or more DSLs and



Properties — Intro Screen Screen

Audio	
Music Asset Name	mscIntro
Music Behavior	StartNew
Play Music In Loop	True
Flow	
Flow Role	Start
Graphics	
Background Picture Asset Name	bgIntro
Background Scrolling Direction	Horizontal
Background Scrolling Speed	3
Misc	
Name	Intro Screen

Properties — FighterBullet Item

Collision Shape	Circle
Description	Bullets fired by the player's
Max Hit Points	1
Name	FighterBullet
Should Flip Horizontal	True
Should Flip Vertical	False
Velocity	Static (100,0)
Wall Collision Reactions	

FighterBullet mass	0.0
Wall mass	1.0
Elasticity	1.0

1 Can…neDefinitionDSL', schema validation skipped.
2 Cannot find a schema that defines target namespace 'http://schemas.microsoft.com/dsltools/EntityDSL', schema validation skipped.

FIGURE 3. Some of the ArcadEx domain-specific languages (DSLs): (a) ScreenFlowDsl and (b) EntityDefinitionDsl.

generators for the iteration subdomain. At the end of the first ArcadEx iteration, game developers had a DSL for modeling screen transition behavior, but they still had to implement all other aspects of the game manually. Subsequent iterations then conceived new (or improved existing) DSLs, focusing subdomains such as entity definition, collision interest, background music management, and keyboard-to-gamepad input mapping. Each DSL includes extensibility hooks to enable custom code to complement the ArcadEngine adapter and directly access FlatRedBall or XNA. We integrated the DSLs with the Microsoft Visual Studio development environment (see Figure 3).

Evaluation

Since ArcadEx's inception, we've been collecting data to assess its effectiveness. Initially, games developed with the first version of ArcadEx had 75 percent of their code automatically generated by the SPL. We had to implement the remaining 25 percent as SPL extensions because some subdomains weren't initially automated, such as wall collisions, score-based events, and initialization of entity properties with random values. Once we retrofitted such extensions into subsequent versions of the game SPL, the number approached 100 percent. If new games have unanticipated variability—that is, if they require behaviors not supported as built-in by the game SPL—then the number will drop again.

ArcadEx games are developed in one-fifth to one-fourth of the time required to develop them using the game engine alone. Such results are in line with MDD improvements measured for other areas.[13]

The reduced level of flexibility in the behavior of the generated games as is (with no extensions), due to increased abstraction levels, is the approach's major drawback. However, as opposed to click-n-play tools, extensibility hooks with full development-environment support and integration are provided for custom behaviors.

A valid concern about using SPL techniques in the digital game domain is whether they threaten the generated games' creativity and distinctness. So far, our results actually indicate that automating the routine and error-prone activities in the game development process (the "commonality") let us spend more time and resources on the domain's variability and extension points, contributing to the uniqueness of each title. In fact, game engines have been responsible for myriad creative, unique industrial titles. Similarly, we don't suggest end-to-end game generators; rather, we recommend layering SPLs and DSLs on top of game engines so that the software reuse is more structured, effective, and intuitive.

Our approach doesn't constitute a complete domain-engineering process per se, with a well-defined and comprehensive set of roles, tasks, inputs, and outputs. Moreover, we don't comprehensively evaluate how current generic domain-engineering tasks fit into digital game development, or how the approach compares to click-n-play tools for simpler game domains.

However, given the peculiarities of digital games, automating game domains shouldn't simply employ software engineering techniques as is, in special SPLs, DSLs, and domain engineering. Using a systematic domain-specific development approach streamlined to digital games, game developers and designers can envision and analyze target game domains and bridge the analysis to core assets in a game SPL. Benefits include reduced complexity for consuming game engines, the breakdown of game development tasks into more granular and automatable chunks, the incremental delivery of value for prioritized game subdomains, domain-specific assets tailored to the unique characteristics of the envisioned family of games, and increased confidence that the resulting games comply with the original vision and requirements.

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES; www.ines.org.br), funded by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) and FACEPE (Fundação de Amparo à Ciência e Tecnologia), grants 573964/2008-4 and APQ-1037-1.03/08.

ABOUT THE AUTHORS

ANDRE W.B. FURTADO is a software engineer at Microsoft and a PhD candidate in computer science at the Federal University of Pernambuco, Brazil. His research interests include digital games development, social technologies, edutainment, and software reuse. Furtado received his MSc in computer science from the Federal University of Pernambuco. Contact him at afurtado@afurtado.net.

ANDRE L.M. SANTOS is an associate professor in the Center of Informatics at Federal University of Pernambuco, Brazil. His research interests include software development for mobile devices, functional programming, compilers, and domain-specific languages. Santos received his PhD in computer science from the University of Glasgow. He's a member of the ACM. Contact him at alms@cin.ufpe.br.

GEBER L. RAMALHO is an assistant professor at the Federal University of Pernambuco. His main research interest is in digital entertainment, including areas such as computer games, artificial intelligence, and computer music. Ramalho received his PhD in computer science from the University of Paris VI. Contact him at glr@cin.ufpe.br.

EDUARDO SANTANA DE ALMEIDA is an assistant professor in the computer science department at the Federal University of Bahia, Brazil, and head of the Reuse in Software Engineering (RiSE) Labs. His research interests include methods, processes, metrics, and tools to develop reusable software. Almeida received his PhD in computer science from the Federal University of Pernambuco. He's a member of the ACM and IEEE Computer Society. Contact him at esa@dcc.ufba.br.

References

1. P. Clements and L.M. Northrop, *Software Product Lines: Practices and Patterns*, Addison Wesley, 2001.
2. J. Blow, "Game Development: Harder Than You Think," *ACM Queue*, vol. 1, no. 10, 2004, pp. 28–37.
3. E.M. Reyno and G.A.C. Cubel, "Model-Driven Game Development: 2D Platform Game Prototyping," *Proc. Game-On 2008, 9th Int'l Conf. Intelligent Games and Simulation*, EUROSIS, 2008, pp. 5–7.
4. E. Folmer, "Component Based Game Development: A Solution to Escalating Costs and Expanding Deadlines?" *Proc. 10th Int'l ACM SIGSOFT Symp. Component-Based Software Eng.*, Springer, 2007, pp. 66–73.
5. A. Lindley, "Game Taxonomies: A High Level Framework for Game Analysis and Design," GamaSutra.com, 3 Oct. 2003; http://bit.ly/2oGHtN.
6. D. Roberts and R. Johnson, "Patterns for Evolving Frameworks," *Pattern Languages of Program Design 3*, Addison-Wesley, 1997, pp. 471–486.
7. R. Prieto-Diaz, "Domain Analysis: An Introduction," *Proc. ACM SIGSOFT Software Eng. Notes*, vol. 15, no. 2, 1990, pp. 47–54.
8. K. Kang et al., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, tech. report CMU/SEI-90TR-21, Software Eng. Inst., Carnegie Mellon Univ., 1990.
9. F. Zwicky, "Morphological Astronomy," *The Observatory*, vol. 68, no. 845, 1948, pp. 121–143.
10. D. Lucrédio et al., "Performing Domain Analysis for Model-Driven Software Reuse," *Proc. 10th Int'l Conf. Software Reuse*, Springer-Verlag, 2008, pp. 200–211.
11. S. Cook et al., *Domain-Specific Development with Visual Studio DSL Tools*, Addison-Wesley Professional, 2007.
12. M. Völter and I. Groher, "Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development," *Proc. Software Product Line Conf.*, IEEE CS Press, 2007, pp. 233–242.
13. S. Kelly, "Domain-Specific Modeling: MDD that Works," blog, 17 Mar. 2010; http://bit.ly/g1KyWp.