JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 7, September-October 2006

Game Development using Design-by-Contract

Richard F. Paige, Department of Computer Science, University of York, UK
Triston S. Attridge, Lionhead Games, UK
Phillip J. Brooke, School of Computing, University of Teesside, UK

Abstract

There are some application domains to which it appears intrinsically challenging to introduce the services offered by formal engineering methods. This paper is an evidence-based presentation that lightweight formal methods are effective in building realistic networked multiplayer games. The evidence is produced via a pilot study that uses Design-by-Contract, under realistic game development conditions, and encompasses both qualitative and empirical results.

1 INTRODUCTION

There are some application domains and development environments that are challenging for introducing the use of formal engineering methods. These include domains such as GUI-intensive systems, consumable software (e.g., Word processing suites) rapid prototyping environments, systems undergoing rapid and unpredictable changes in requirements, and large-scale distributed systems, where non-functional requirements can make it difficult to use formal techniques. In order to make a strong case to developers and managers in these and other domains, evidence must be presented that formal engineering methods are usable and effective in these domains. Such evidence must be both qualitative and empirical, in order to produce a coherent argument, and must target the specific contributions that formal engineering methods can make to the specific development problems experienced in these domains.

In this paper, our agenda is to focus on one specific application domain, networked multiplayer games, and to demonstrate qualitatively and empirically that lightweight formal engineering methods can be usefully and effectively applied in their construction. This is an important domain on which to focus: it is financially significant, offers substantial technical challenges, and there is evidence of a desirein this industry to explore the use of rigorous software engineering practices.

The game development industry is growing in size and maturity. Projects now command budgets in the millions of pounds, with large numbers of developers working for years [1]. As the complexity and cost involved with developing a commercial computer game increases, there has also been an increased drive by managers in the field to adopt good software engineering practices, such as formal engineering methods, to achieve greater productivity and in order to release high-quality code more quickly. However, no concrete evidence has been provided that formal engineering methods can solve some of the specific problems of game development, particularly early defect detection, enabling reuse of components, and helping to track down bugs more rapidly. Moreover, there are complaints of a lack of compelling evidence that formal engineering methods can fit into commonly used game development processes. Such processes are, within the industry, described as organic, with overwhelming need for rapid feedback cycles [21,22] to customers and financiers. At first this appears to be at-odds with what formal engineering methods can provide.

This paper takes an evidence-based approach to demonstrating the value of formal engineering methods to the game development industry. It reports, qualitatively and quantitatively, on an experiment in the use of the design-by-contract (DbC) [18] lightweight formal method for building a realistic networked, multiplayer game. It is possible to argue that DbC is an ideal candidate for introduction to game development because it is a executable and lightweight formal engineering method – writing contracts is the same as writing code – and as such the technique appears to be compatible with the general practices of game developers. What remains is to provide qualitative and quantitative experimental evidence for game developers and project managers that DbC has benefits.

To this end, we present an initial study in developing a networked multiplayer game; we see this work as the first step in providing detailed and broad empirical evidence of the utility of formal methods in this domain. Thus, this paper presents a feasibility study in order to help persuade the game development industry that there is value in carrying out additional pilot studies in the use of formal engineering methods.

The initial study was carried out under simulated industrial game development conditions, following a typical process. The experimental hypotheses, method, and quantitative analyses are presented in Section 5. Qualitative assessment is presented throughout the paper, particularly in the conclusions.

We commence with a brief overview of a prototypical game development process and clarify core problems in game development that could be attacked using formal engineering methods. We give a short introduction to DbC and the flavor that we applied during the experiment, including examples of formal specifications. We then briefly describe the multiplayer game that we constructed, focusing on its features, before outlining its abstract architecture and design. The rest of the paper presents experimental evidence from the development stages, where we discuss several categories of contracts, how they were applied, and how they correspond to the experimental hypotheses. We analyze the results, and use them to evaluate the architecture and game design process. We then conclude and discuss several lessons that were learned during the experiment.

2 GAME DEVELOPMENT AND DESIGN

Computer games are created by studios that employ groups of full-time designers, programmers, and artists. Personnel are divided into teams, each focusing on one product. Games are typically financed by publishers, in return for a royalty package.

Game development is substantially risky: it is considered very difficult to predict whether a new product will be a success or an abject failure [7].

The development of a game has a lifecycle of years and typically involves teams of over 20 people. Such projects have tight schedules, with concrete milestones – e.g., showing the publisher a prototype in order to justify receiving the next advance of funds. Many tasks during the project lifecycle require external review, e.g., by the studio head, project leader, or publisher.

Design documentation is used during game development; it is a key tool for (non-technical) game designers, but is not looked favorably on by technical designers and programmers. Documentation ranges from high-concept statements [22] which are often used to help obtain advance funding for a project from publishers, to online documentation via Wikis for exploring game scenarios, storyboards, and game play specifications. There is some advocacy for the use of assertions as a documentation technique but no documented use of formal engineering methods beyond assertions in code. The typical use of assertions in game development [19] is to notify programmers of problems that arise during dynamic loading of game assets (e.g., art, sound, other data). Assertions are therefore used more for sanity checks than as a formal method for documenting contracts between clients and suppliers, as recommended by the DbC approach.

There is no standard game development process, but variants of the spiral model [5], evolutionary prototype [21], Tiered Development [21], Data-Driven Design [20] and Extreme Programming [4] have been applied with varying degrees of success. What these approaches all have in common are the following:

- An iterative approach to development that allows continual improvements to be made while minimizing the impact on scheduling.
- Decomposition of a game play specification – which outlines how one or more players interact with the game – into sections that correspond to iterations.
- Emphasis on the implementation and testing stages of the engineering process.
- Component and architecture reuse from previous developments. A typical example is to reuse and extend a game engine, e.g., Quake and Unreal [9,11].

The basic activities of game development are therefore as follows:

- Construct a game design that captures the requirements for the game itself; this involves constructing stories, worlds, characters, and scenarios for game play.
- Construct a game play specification that will drive the technical development.
- Develop the game based on a decomposition of the game play specification, focusing substantially on the implementation (coding) and testing phases.

Many games – particularly networked ones – are emergent systems; moreover, game play is an emergent property, based on complex interactions among different rules. Emergence arises from the user-game relationship. This makes it difficult to detect defects and isolate them in the code, and is one of the reasons why we think that lightweight formal engineering methods will prove profitable in this domain.

We can now identify key requirements for any formal method that is a candidate for use in game development:

- It must work with both legacy components and new components, and enables reuse of components that it produces, e.g., via improved interface specification.
- Iterative and incremental development is supported.
- It places minimal emphasis on the production of documentation during technical design and development.
- It is more cost-effective in diagnosing defects than existing techniques.
- It helps identify incorrect design decisions more easily than existing techniques.
- It makes it easier to track down bugs in the source code than existing techniques.

For some of these points, we will present empirical evidence that supports the case for using DbC for game development; for other points, we will argue qualitatively. For all points, we base our presentation on the development of a real system.

3 DESIGN-BY-CONTRACT

Design-by-contract (DbC) is a lightweight formal method that aims to increase code reliability. Meyer [18] explains how the mechanism fits within object oriented software construction. The central idea is to view the relationship between a class and its clients as a formal agreement framed as a set of rules. If a client promises to only use the class under certain circumstances, then they can be certain that the class will carry out a task. If the client uses the class in the wrong circumstances, then the class's behavior is undefined - it can do anything it likes.

A contract is normally checked at run-time by evaluating assertions on method invocation: clients check preconditions, and method bodies check postconditions. Additionally, classes may have an invariant, which represents the properties that must be preserved by all methods [17]; these are checked both before and after method invocation. Contracts have several benefits [18]:

- They can help to capture otherwise implicit assumptions about a system's architecture and its components.
- They can make it easier to carry out verification, validation, reuse, and the construction of robust components and systems, which will thus enable easier incremental change.
- They can enable reuse of components and architectures, by capturing conditions that can affect the correct operation of software, and by embedding these conditions in the software directly.
- They can enable automated testing and static analysis of system properties.

These expectations, and the problems associated with detecting defects in computer games, make DbC a good candidate to apply in game development.

In our experiment, we made use of C++ as the implementation language, in which DbC was applied. C++ is a standard language used in game development, though it is typically restricted so that multiple implementation inheritance and the Standard Template Library are avoided (due to their associated performance penalties). Contracts were implemented using the C++ preprocessor and a contract library; this made it easy to turn contracts on and off at compile time. We used pre- and postconditions of methods at the implementation level in the experiment. Invariants were integrated with pre- and postconditions. We give several examples of contracts in the sequel.

4 THE GAME SPECIFICATION AND ARCHITECTURE

The game we produced was called *Contract Princes*, a real-time strategy game. Players cooperate to build up an economy in a post-apocalyptic future. Players may acquire, trade, and manage multiple resources while defending their society against incursions from barbarians and other players. The game supports multiple players only; however, there are internal (non-player) artificial intelligences acting in the game in order to provide additional trading partners. The visual scheme is based on an amalgamation of 3D shapes, from the viewpoint of a zoomed-out, isometric, top-down view over the game terrain. Within the game, there are units (controllable entities that can be given orders), buildings, and meta-objects (which can be interacted with but not given orders). Orders are generated using a mouse interface with shortcut keys, but tactics, which encapsulate orders, can also be defined and saved. Squads of units can be defined and orders directed to an entire unit; controlling a large number of units may involve a performance penalty.

The game was developed by following a variant of data-driven design [20]; commercial games that have been constructed using this process include Total Annihilation [6] and System Shock [12]. The idea behind data-driven design is to separate data used by the system from the code, so that recompilation is not necessary when changing data. Tailoring the game then involves tailoring the data.

Architecture and Design

Contract Princes has a client-side and server-side architecture. This architecture is shown below. The player sends order requests to the server, which carries out a confirmation process. Confirmation is returned to the client, which goes through a process of altering client behavior – in terms of plans. These plans spawn a number of commands that in turn alter the internal subsystems of the client. The internal subsystems include a *property system* as well as a path finding system, map system, and a blocking node. The game engine makes use of the internal subsystems as well.

Objects in the game world – including both buildings and units that accept commands – have properties (e.g., the object can be moved, can be used as a weapon). This is managed by an archetype property system. Archetypical objects in the game are assigned properties. These objects are also placed in a catalogue, as instances of objects that behave in the same way. Objects in the catalogue can inherit properties from each other, and new objects can be defined that make use of existing properties. This approach allows the base properties of objects to be changed quickly, with the changes propagating to all relevant objects quickly. The system allows assignment of properties to objects, and management of how objects inherit from each other. A user interface is provided on the client-side for managing properties. There is substantial complexity in removing properties when a chain of inherited properties exists; this is documented in [3].

The network component to the game is event-locked rather than frame locked to improve response time. The game server broadcasts advanced-turn packets at a regular interval. These packets give permission to clients to advance to the next turn. Interspersed with these are user command requests authorized by the server. These are broadcast to all clients if they pass the validation process.

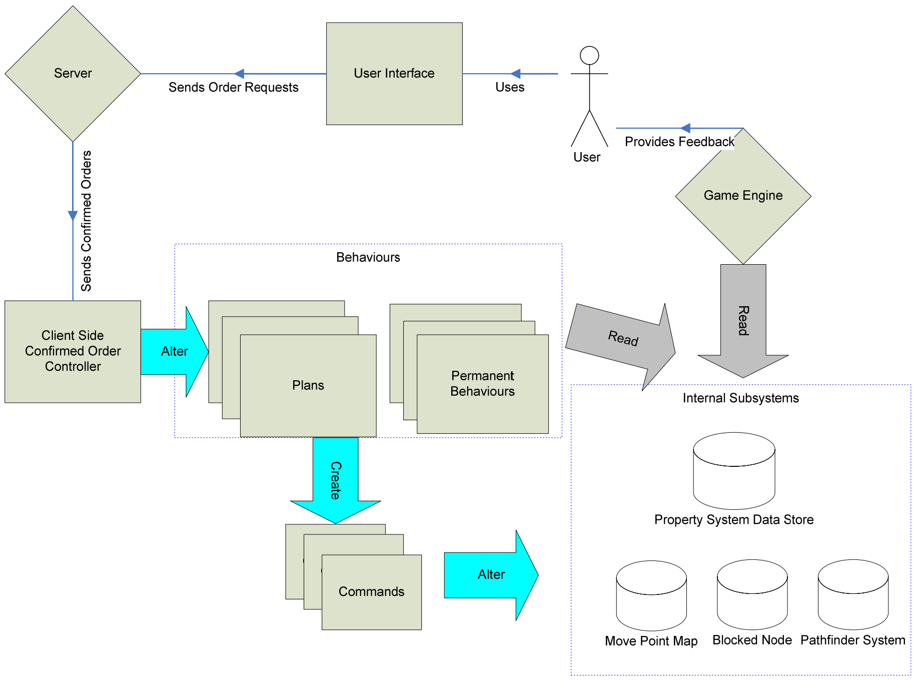

Fig. 1: Overview of Architecture

The two remaining components in the design are the plan abstraction and the game engine; we refer the reader to [3] for discussion on the game engine: we reused the open-source Crystal Space engine. Plans represent what a unit is intending to do over the next few game turns; a plan persists from turn-to-turn. When a confirmed order arrives from the server, a plan for a unit or set of units is changed. Processing a plan eventually results in game state changes that execute the plan. Clearly, to retain consistency, changes to game state should occur in the same order on all clients. As well, game state changes must be processed in a batch, and plans must be independent, i.e., executing one plan cannot affect another.

Plans are designed using the Command design pattern; they encapsulate commands – i.e., all game state changing actions.

5 QUANTITATIVE EVIDENCE AND DISCUSSION

We now set the stage for our presentation of quantitative evidence to support the use of DbC in game development. In particular, we develop three hypotheses, present arguments for their validity, and discuss the experimental method (and its limitations and caveats) that we applied.

One of the goals of DbC is to explicitly capture implicit assumptions about system architecture, via contracts on classes and methods. In particular, this should be useful for decreasing debugging time in games where dynamic loading takes place (e.g., assets loaded at run-time), since well-formedness constraints on assets can be captured explicitly via a contract.

Additionally, defects in games are arguably harder to track down than in typical information systems where DbC has been successful in the past. The main focus in a game is the user-game interface, where interactions across the interface are defined by emergent behavior evolving from a small set of simple rules [23]. Emergence, coupled with the varied environments that games operate in, plus multiplayer network capabilities, make it difficult to trace, reproduce, and distinguish defects from correct behavior. Contracts capturing such emergent behaviour would be desirable. However, these contracts – termed synchronisation contracts by Beugnard [24] – are generally not supported by lightweight contract mechanisms since they involve modelling propositions over sequences of states, and these are expensive (or even impossible) to check without using theorem proving. Such a mechanism would be incompatible with the requirements of game development and thus we will restrict ourselves to partially capturing emergent properties via pre- and postconditions. Examples follow in the sequel.

The following hypotheses were formed for the purposes of experiment:

1. Contracts are useful for efficiently diagnosing defects that arise during the implementation phase of game development. The measure of utility shall be that contracts help find more defects than traditional techniques – unit testing, code reviews, compilation and static analysis, etc – and do so in what is estimated to be a more timely manner, based on prior experience
2. Contracts will reveal defects that are otherwise difficult to find in technical game development, thus decreasing test time.
3. Contracts will reveal broken assumptions that are made during game technical design but that are invalidated by the code.

Experimental Method

Since all the hypotheses related to diagnosing defects, the data collected in the experiment focused on defects, particularly where they were found, and the type of error. These were recorded in a detailed error log. Static errors were caught by the compiler and do not help with the hypotheses. Run-time errors were detected via contract failures, testing, or automated checks that were encoded in the program itself. Defects associated with run-time errors were of most use in helping check the hypotheses.

Hypothesis (1) was tested by recording the number of defects caught by contract failures, versus those caught by other means (e.g., the compiler). Additionally, the level of assistance that a contract provided to the debugging process could be qualitatively gauged. There are different types of defects that arise [2]; the defects discovered were placed into categories depending on how quickly it was possible to track down the defect.

Hypothesis (2) was tested by collecting data on relative testing times used for solving difficult defects, and from this extrapolating how difficult it would have been to form a contract to catch each defect, given our understanding of the cause of the defect. Ideally, we would want to use two groups to analyze this hypothesis: hard-to-find defects would be added to a code sample, with one test group acting as the control, and the second adding pre- and postconditions before testing. Time constraints prevented us from taking this approach, but we plan to complete this in a separate experiment, and we make some qualitative observations in the next section.

Hypothesis (3) was tested by including the type of error in each run-time error report. As with tracking categories, error categories were used as well. If a significant proportion of contract failures found broken assumptions then we considered this hypothesis to be proved.

An overarching issue in the experimental method is in the comparison of using DbC versus not using it. To show an advantage in the former, we would need to repeat the experiment without contracts. This was not possible due to time and financial constraints. As well, we would only be able to gather coarse-grained statistics, e.g., overall development time, defects found, making a fine-grained comparison in terms of the three hypotheses impossible (we discuss this more in the next section). An alternative would be to collect data over several projects and synthesize the results. However, it was possible to *locally* contrast the use and non-use of contracts. There were cases where contracts were omitted in the implementation, and comparing the details of errors that were found by not using contracts will provide some primitive and inconclusive data.

As part of the method, a detailed error log was kept that, for each error, recorded:

- An explanation of what the error was and where it occurred, as well as the type of error: logic error, error in implementation, invalid assumption, misunderstanding a reusable component (e.g., in Crystal Space), language error (e.g., function hiding in C++), or flawed contract.
- How the error was caught: precondition or postcondition failure, testing, run-time error handling mechanisms (e.g., fatal crash, divide-by-zero).
- How hard the error was to locate: easy (instantly obvious by looking at the code), medium (debugging necessary using the call stack and variable values), hard (extensive debugging work beyond call stack tracing), and unresolved.
- The action taken to fix the error: quick fix (changing a line or small section of code), function refactoring, class refactoring, system refactoring, no fix made.
- For errors not caught by contracts, how easy it would have been to craft a contract that would have detected the problem.

Analysis of the Implementation

Implementation took approximately 45 days full-time and led to source code consisting of 400 files. A summary of the findings is in Figs. 3-7, below. During implementation, 80 semantic errors were detected, with 49 of these detected by contract failures; precondition failures were in the substantial majority. The errors were roughly evenly distributed amongst easy, medium-, and hard difficulty for resolution, and most of the errors were due to logic and implementation errors. Perhaps surprisingly, only 5% of the errors were due to flawed contracts. Also surprising was the fact that 62% of the errors were repaired by quick fixes that took on the order of minutes. We also observed that errors not caught by contracts would have been difficult to catch by a contract. We now discuss how these findings correspond to the individual hypotheses.

Contracts will be useful in diagnosing defects

The data summarized in Figs. 3-7 contained in our logs, suggest – not surprisingly – that contracts help in diagnosing defects. A high percentage of errors were caught by contracts. Fig. 8 and 9 compare locational difficulty with method of tracking, in both absolute and weighted terms. Fig. 8 shows an increase in the effectiveness of tracking if errors were caught by a contract. This is more clear in Fig. 9 where it is shown that the pre- and postcondition tracking methods have much higher ratios in the easy-to-difficult categories compared with testing or run-time errors.

The comparison shows that errors caught by postconditions generally resulted in easier tracking; this makes sense as a postcondition is restricted to checking that a method has kept their side of the contract, and since most methods in a good OO design are small, the occurrence of the error should be close to the contract.

Preconditions have a larger ratio of ‘hard’ to ‘medium’ difficulty than postconditions. This is counter-intuitive, as a precondition failure should indicate an error in the client code. This is partly due to the event-driven nature of games: a precondition failure was often not due to a client error but a predecessor in the chain of calls. In theory, these errors should have been caught by an earlier postcondition, but in practice preconditions were often acting as “postconditions-by-proxy”.

Contracts Reveal Otherwise Hard-to-find Defects

The data obtained did not support this hypothesis. The hypothesis was originally phrased in terms of whether a contract could have been constructed to catch an error that wasn’t otherwise caught by a contract. The data in Figs. 3 through 7 suggests that errors that aren’t caught by contracts are not in general easy to write contracts to catch. This is not entirely identical to the hypothesis but it is at odds with it. Our explanation comes from Fig. 3; this suggests that most errors were caught by contract failures of some kind; any resulting errors not caught by contracts are probably going to be difficult to express as contracts, since numerous other specifications (including tests) did not catch the problem. The error log does support this. Errors that were difficult to catch using contracts were generally caused by component misunderstandings or language misuse, and these are typically very hard to deal with via contracts. Thus, given the experimental data, we must reject our initial hypothesis.

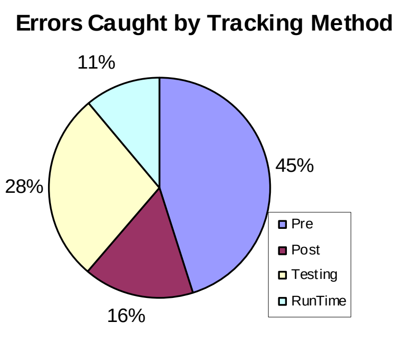

Fig. 3: Errors Caught by Tracking Method

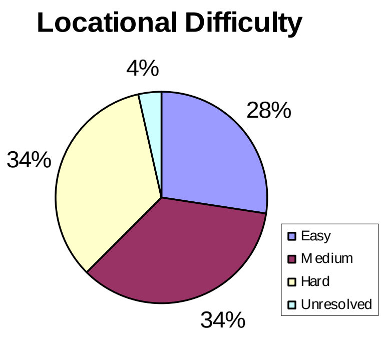

Fig. 4: Locational Difficulty of Errors

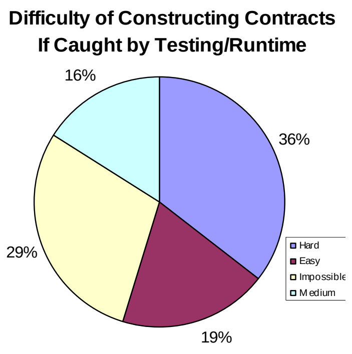

Fig. 5: Estimated Difficulty of Constructing Contract after the Fact

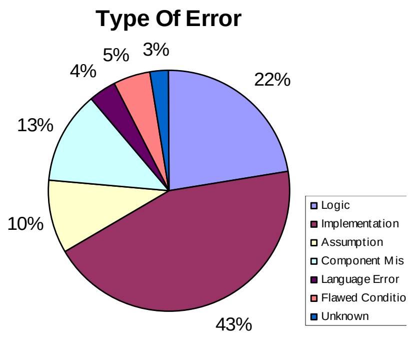

Fig. 6: Causes of Error

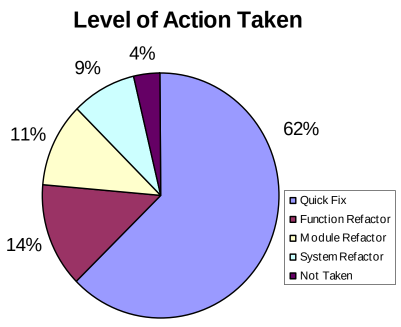

Fig. 7: Level of Action Taken

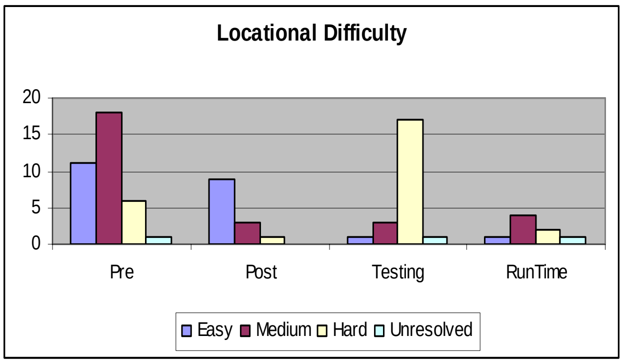

Fig. 8: Comparison of Locational Difficulty with Tracking Method

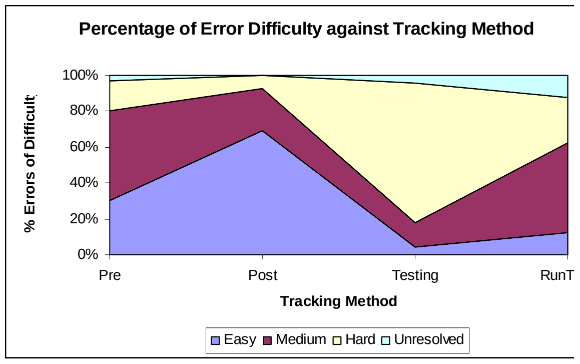

Fig. 9: Weighted Comparison of Tracking Method vs Locational Difficulty

Contracts will reveal broken assumptions

The data supports this hypothesis – contracts were useful in reveal broken assumptions. However, not all broken assumptions were caught by contract violations. An analysis of the weighted error types of each tracking method is contained in Fig. 10. Precondition, postcondition, and testing methods revealed roughly the same percentage of broken assumptions.

The low overall number of assumption failures is logical in hindsight. The original hypothesis assumed that in a rapidly developing project such as a game, programmers would likely not keep track of a change in how a class is used. As the code was predominantly implemented by one person there were fewer assumptions to be communicated. This suggests that the best way to test this hypothesis is to repeat the experiment in a multi-person project

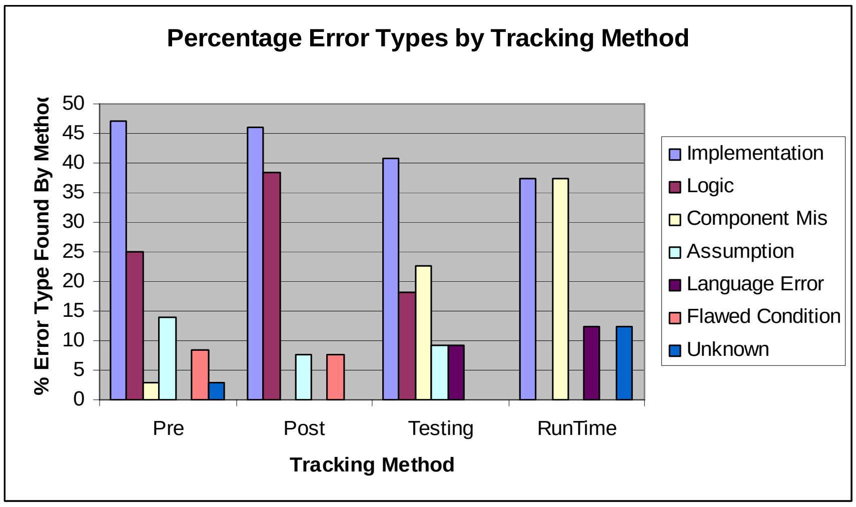

Fig. 10: Error types by Tracking Method

There is an interesting example of a broken assumption that is worth further discussion. The general approach followed when an error occurs was to write a test to collect more data; this leads to being able to detect where the error arises. Contracts are then reinforced in this area so that the error is either caught by a contract failure, or the programmer can conclude that the error is located elsewhere. Thus, contracts are used as a direct debugging mechanism. This approach was very useful in analyzing the complicated list-checking functionality of the path finding algorithm in the game; the pathfinder was returning unexpected paths, and contracts were useful in determining why. This is a good example of contracts helping to track defects due to emergence in games. However, there is a risk with using this approach. A particular example also arose with the pathfinder, which was not returning the shortest path between two points. Contracts were repeatedly strengthened but the problem could not be diagnosed. It turned out that the expected error did not exist: the code was functioning as designed – the design was incorrect. It would be useful to apply contract simulation during design to help catch this admittedly non-trivial problem.

6 DISCUSSION

Recall our requirements for game development listed earlier: in particular, a method argued to be useful for game development should support reuse and inclusion of legacy code, support incremental and iterative development, minimise documentation, and better help diagnose errors. The quantitative evidence presented in Section 5 adds to an argument for DbC supporting the last point. As well, it is clear that DbC supports the other requirements as well. DbC is a code-based formal engineering method and thus can be used in any game development process, including data-driven design and Extreme Programming; for particular discussion on the latter, see [25].

Support for legacy is also possible, as argued by Meyer [18]. Moreover, contracts – while a form of documentation – are also code, and as such do not promote the creation of documents that may become inconsistent with code: consistency between contracts and code is guaranteed (otherwise a run-time or compile time error will arise).

As mentioned earlier, our use of contracts was restricted to the implementation phase; this is not an unreasonable restriction as it is the emphasized phase in game development. As well, we primarily used pre- and postconditions in development; class invariants were used implicitly and were automatically added to relevant pre- and postconditions. In part this was because the pre-processor that we were using did not support invariants.

Some examples of contracts follow. In total there were approximately 140 contracts used throughout the code, as well as approximately 100 additional assertions used for "sanity checks" following the advice in [19]. Each contract included a boolean condition (evaluated on entry or exit to a method) and a textual string that was output during testing and debugging should the contract fail at run-time.

The first example is used in the game world class, instances of which represent the overall world in which game play takes place. This precondition states that the list of meshes that make up the game world's data must be empty when attempts are made to synchronize the mesh entities with the world.

```
void      cGameWorld::SynchroniseWithWorld( void )
REQUIRE( m_pxGameWorldData->m_mapMeshList.empty(),
"Should only synchronize if empty!" );
```

The second example demonstrates a postcondition. The game allows the construction of attack plans, which can then be executed by units in the game (e.g., move a tank five miles east, then launch weapons). The method AttemptToMoveCloser attempts to execute part of a plan; on termination, it must be the case that a valid move intention has been constructed.

```
const bool cPlanAttack::AttemptToMoveCloser(...)
ENSURE( !m_boCanMove || m_pxIntentionMove, "We have not got a valid move
intention !" );
```

The third example demonstrates a contract for part of the geometry system. Player movements must be clamped to the geometry of the map for a world. The method ClampToBoundary carries this out but only under the condition that the movement is in bounds, i.e., a ratio measure is between 0 and 1.

```
REQUIRE( fRatioFromBoundary > 0.f && fRatioFromBoundary < 1.0f, "Invalid
ratio!" );
```

Some Observations

We observed that contract failures often occurred in the same part of code; either the same contract would fail repeatedly over time, or a series of contracts in close proximity would fail as a group. Our theory was that this was tightly related to the quality of the underlying code – the code itself was error prone, and therefore could be a good target for refactoring and rework. Patterns of contract failures are in general a good mechanism for assessing the quality of code.

A specific observation can be made about postcondition failures. In several cases we discovered a postcondition failure indicating a discrepancy between how a method was originally thought to work, and how it was implemented. In general, this led us to refactor the architecture.

As the implementation proceeded, it became clear to us that there were circumstances where it was extremely difficult to express a postcondition succinctly; preconditions were, generally, straightforward to capture. However, it was observed that though a postcondition could not easily capture a constraint, the constraint could be succinctly captured using a unit test.

The pre-processor based mechanism that was used for design-by-contract in C++ was cumbersome, though it was reasonably flexible as it allowed different contract recovery schemes to be implemented. It was also easy to use for turning off all contracts, e.g., for the released version of the code. The main limitation with the pre-processor mechanism was the lack of debugging support; with an integrated contract mechanism, e.g., as in Eiffel, the debugger is aware of contracts and can be used to more quickly track down a failed contract.

There is a risk that the contract mechanism may be incompatible with automated testing techniques that are typically used by gaming companies [16], particularly for regression and integration testing. In an automated testing environment, the priority may be to get as much of the code tested as possible; contract failures may interfere with this. Full automation of testing is an argument for defensive programming; there is a potential compromise, however: exception handlers can be written to deal with failed contracts, and certain at-risk interfaces could perhaps be annotated with defensive programming-style checks, in order to automate the testing process. However, exception handlers are not widely used, nor looked upon favourably by game developers.

The use of a pre-processor mechanism led to increased compile times and execution times. For example, for preconditions, there were common checks included in header files (e.g., that parameter values were in range); these header files were included in many places, thus slowing down compile and execution times. However, our experiments suggest that the net increase in compile time is small, and contracts can be turned off in release code in order to obtain the highest performance.

A large proportion of the most difficult-to-track errors were caused by component misunderstandings, for example, how a component in Crystal Space behaved. Erroneous assumptions of how such components behaved were generally difficult to diagnose; contracts were of limited help because these reused libraries did not make use of them. Contract wrappers could be used here, particularly for a framework like Crystal Space, which is large and complex and – in some cases – poorly documented.

The game architecture included a number of architecture-level constraints, e.g., for keeping the game state consistent between multiple clients. Subsystems were implemented with these constraints in mind, but the constraints were not formally captured using contracts. It would be useful to express contracts in the design that would check these constraints. For example, the game implementation included unit behaviors; a behavior is functionality that is applied to all objects satisfying a query. The method that implements this feature was not captured in the original design. Behaviors were therefore initially implemented in a naive way, and directly modified object properties without routing through suitable commands. This violated the implicit architectural constraint, but it took a substantial amount of time to notice this and deduce the core problem, because this behaviour only emerged via checking a large number of user-game interactions. Contracts partly helped in diagnosing the problem, but they did not support direct checking of the constraint, because we could not easily encode the desired behaviour in a contract; moreover, we would argue that it would be difficult and undesirable to require game developers to do so. In this particular case, a good indicator of the problem would have been a warning that an external system was being modified before a synchronization point was reached. Typically such checks are made at run-time, but there is the potential to make this a compile-time check via splitting read/write sections of class interfaces into two parts, and then using a meta-programming mechanism. In either case, explicitly enforcing logical and physical dependencies in the architecture would be advantageous. This would require a clear architectural specification; in a game where the architecture changes dramatically (e.g., Black and White, when the architecture was modified substantially to add multiplayer mode) capturing, maintaining, and checking these contracts would be much more difficult.

7 CONCLUSIONS

Our investigation into the use of DbC for game development suggests that the approach is useful and productive, particularly for diagnosing and tracking down defects; run-time errors that were caught by a contract failure were significantly easier to track down than those detected by other means. Although evidence to support our second hypothesis – that contracts can help to reveal hard-to-catch problems – was not immediately forthcoming, the experiment suggested that an individual contract tends to have a wider scope of applicability than originally expected. This is related to the domain of application and the nature of distributed code. Contracts also showed use in detecting broken assumptions, and this is clearly an area for future work, particularly if the techniques can be applied earlier in the game development process – e.g., during design or architecting. The challenge here is to identify a form of architectural or design model that appeals to game developers. Rough or whiteboard sketches of UML models have appeal but do not lend themselves to rigorous analysis.

We expect that some of the conclusions that we have drawn are applicable to distributed systems in general. The fact that DbC is a scalable technique thus makes it an attractive general-purpose formal engineering method for use in this broader domain.

The use of contracts did not place any noticeable delay on producing the working game. In part this was because the number of contracts written was not unduly large, especially compared to the overall system structure. We also estimate that the use of contracts reduces the overall amount of time spent on testing and debugging; the empirical evidence we have indicates a reduction in time for finding and eliminating bugs; additional experiments must be carried out to assess the overall effect of using DbC on testing.

Additional experiments would be useful and should be carried out, particularly focusing on larger teams, and on the use of contract failures as predictors of flaws in code. As well, it would be useful to determine a mechanism for separating contract failures into ones due to local (in-method) defects and those due to chains of calls. The fact that this work concentrated on the implementation stage – a realistic focus, given the nature of game development – suggests that there may be benefits to exploring the use of design-by-contract in earlier stages, particularly the aforementioned architectural constraints

REFERENCES

[1] Arey, D. "Post Mortem: Naughty Dog's Jak II", Game Developer Magazine January 2004,

[2] Astels, D. "Test Driven Development: A Practical Guide", Prentice Hall (2003)

[3] Attridge, T. Design-by-Contract for Game Development Project Site. http://www.cs.york.ac.uk/~paige

[4] Beck, K. "Extreme Programming Explained: Embracing Change". Addison Wesley (1999)

[5] Boehm, B. A Spiral Model of Software Development and Enhancement, IEEE Computer, May 1988.

[6] Cavedog Entertainment. Total Annihilation, Cavedog Entertainment (1997)

[7] Crawford, C. "Chris Crawford on Game Design", New Riders Publishing (2003).

[8] Crystalspace SourceForge Project. http://crystal.sourceforge.net/

[9] Epic MegaGames. Unreal Tournament, GT Interactive (1999)

[10] Falstein, N. "Better By Design: The 400 Project", Game Developer Magazine 9(3) (2002).

[11] id Software. Quake III Arena, Activision Publishing, Inc.(1999)

[12] Irrational Games & Looking Glass Studios. System Shock II, Electronic Arts (1999)

[13] Kreimeier, B. "The Case For Game Design Patterns", www.gamasutra.com (2002)

[14] Leonard, T. “Postmortem: Looking Glass’s Thief: The Dark Project”, Game Developer Magazine 6(7), 1999.

[15] Looking Glass Studios. Thief: The Dark Project, Eidos (1994)

[16] Maxis Software Inc. The Sims, Electronic Arts (1999)

[17] McConnell, S. “Code Complete”, Microsoft Press, 1993.

[18] Meyer, B. “Object-Oriented Software Construction”, Second Edition, Prentice Hall (1997)

[19] Rabin, S. “Squeezing More Out of Assert”, Game Programming Gems, Charles River Media (2000)

[20] Rabin, S. “The Magic of Data-Driven Design”, Game Programming Gems, Charles River Media (2000).

[21] Rollings, A. and Morris, D. “Game Architecture and Design”, The Coriolis Group, (2000)

[22] Rollings, A. and Adams, E. “Andrew Rollings and Earnest Adams on Game Design”, New Riders (2003)

[23] Zimmerman, E. and Sallan, K. “Rules of Play: Game Design Fundamentals”, MIT Press, (2003).

[24] Beugnard, A., Jezequel, J.-M. , Plouzeau, N., and Watkins, D. “Making components contract aware”. IEEE Computer 32(7) (1999).

[25] Paige, R.F. and Ostroff, J.S. “Specification-Driven Design for Teaching Lightweight Formal Methods”. Proc. Teaching Formal Methods 2004, LNCS 3294, Springer-Verlag, November 2004.

About the authors

Richard Paige is a lecturer at the University of York, United Kingdom, where he works with the High-Integrity Systems Group and is a co-leader of the Software and Systems Modelling Team. He completed his PhD in Computer Science at the University of Toronto in 1997.

Triston Attridge is a game developer at Lionhead Games. He completed his degree at the University of York in Computer Science in 2004.

Phil Brooke is a principal lecturer in the School of Computing, University of Teesside, United Kingdom, where he works on dependable systems, formal methods, and object-oriented systems. He completed his DPhil in Computer Science at the University of York in 1999