

Introducing XP into Greenfield Projects: Lessons Learned

Jonathan Rasmusson, *ThoughtWorks*

One of ThoughtWorks' clients, TransCanada Pipelines Limited, is a leading North American energy company that specializes in natural gas transmission and power generation. TCPL builds many custom applications using Sun Microsystems' Java J2EE. TCPL engaged ThoughtWorks for two reasons. First, TCPL wanted to determine whether Web service technologies and standards were mature enough to warrant building applications within the enterprise. Second, they wanted us

to rewrite from scratch a legacy application to demonstrate the technology's ability to integrate with other applications. The legacy application was an event notification system written in Forte and Sybase procedures. By writing custom applications monitoring specific business events (for instance, the price of natural gas), users could use the event notification application to notify interested parties.

When ThoughtWorks recommended Extreme Programming¹ as a development approach, some of TCPL's architects and project

managers had concerns. For instance, XP's lack of investment in up-front design and documentation differed from the more traditional forms of application development they knew. They also weren't sure who would be responsible for the system's overall architecture. Fortunately, converting skeptics was not a task for the project team. Rather, we aimed to demonstrate that XP worked by example. We had six months to prove the technology, train the team in XP, learn the business, and produce a production-ready application.

The project team comprised approximately 13 people, including a project manager, an iteration manager, a build master, a business analyst, a tester, four junior developers (with no prior XP experience), and four intermediate to senior developers (who had XP experience). Upon delivery, the greenfield code base

ThoughtWorks introduced Extreme Programming into an organization and successfully completed a bleeding-edge technology project with client staff that had no previous experience using an agile development approach. This article describes how XP helped the project succeed and shares lessons learned.

XP's social side really shone through when people got together, shared experiences, and collectively looked for solutions.

had 21,000 lines of application code and 16,000 of test code.

The practices

The project applied many software development practices—not all of them exclusive to XP. The following practices worked well and contributed significantly to the project's overall success.

Pair programming

I believe pairing is the most powerful XP practice and made the largest contribution to our overall project success.² Unit testing, refactoring, test-first design, and simplicity are vital. Pairing, however, let the team apply these practices to their fullest extent. Pairing was the most effective way to communicate and demonstrate XP practices to new team members. Not unexpectedly, people embraced XP at different speeds. Not everyone immediately started writing test-first code, it took time to learn effective unit testing, and refactoring was not second nature. Pairing, however, brought a level of consistency to the team regarding XP practices.

Advice for senior developers pairing with junior developers would be to work at the junior's speed. A junior developer commented that he initially found pairing difficult because the senior person would work too quickly. The junior felt more like a passenger traveling on a high-speed train while the senior developer did all the work. Both developers in a pair should feel like they're contributing as effectively as possible.

Stand-up meetings

Fred Brooks reminds us of the importance of communication when describing why the Tower of Babel failed.³ The Tower of Babel was man's second major undertaking, after Noah's Ark. This project had almost everything going for it: a clear mission, plenty of manpower, materials, time, and sound technology. So, why then did a project with so much going for it fail? It lacked communication and organization.

XP advocates daily stand-up meetings as a means of enhancing project communication. Just putting everyone in a room and quickly sharing knowledge at a high level efficiently enables mass communication between team members and stakeholders. Team members ranked stand-ups very high among factors

they considered essential to the project's success. They became "town hall meetings" that everyone wanted to attend for updates on all the daily project events.

The trick to holding effective stand-ups is to set expectations early. Let the team know that the point of the meeting is to communicate quickly and clearly anything directly related to the project. Design issues, requests for help, and problems and solutions discovered are all relevant. Meetings should typically not exceed 15 minutes, and attendance is mandatory.

Other positive side effects of stand-up meetings included project visibility and strong team building. Everyone from developer to project manager knew the project's state at any given time. You could raise small problems and address them before they became big problems. Also, XP's social side really shone through when people got together, shared experiences, and collectively looked for solutions.

Unit testing

If I could only recommend one coding practice to software developers, those who use XP or otherwise, it would be to write unit tests.

Unit tests are low-level tests developers write (typically at the class method level) to prove that their code behaves as expected. On this project, these tests were invaluable in letting us refactor.⁴ The tests provided immediate feedback when we broke something in the code. They highlighted the sometimes subtle and difficult-to-detect differences between our production and development environments. In a sea of change, the unit tests were our anchor.

Everyone on the team bought into the concept of unit tests and saw their value. Effective unit testing, however, took time to learn and was best implemented in pairs. By pairing those with JUnit (www.junit.org/index.htm) experience and those new to unit testing, we quickly got everyone into the habit of writing tests.

Developers seem to evolve somewhat when it comes to writing unit tests. The first stage, becoming *test infected* (<http://junit.sourceforge.net/doc/testinfected/testing.htm>), is often simply convincing developers that writing tests is a good thing. Once comfortable with the mechanics of unit testing, developers can look at more advanced techniques, such as test-driven development and using MockObjects (www.c2.com/cgi/wiki?MockObject). Eventually, test-infected developers will also start to use tests in

their object-oriented designs. When given a design problem, instead of focusing immediately on the implementation, developers can use the tests to help establish the object's interfaces first. Once the tests are written (what the object is supposed to do), they can go on to make them pass (how the object behaves).

When writing unit tests, watch out for duplication. Although you don't want to be overcritical about it at the project's beginning (after all, we want developers writing lots of tests), you need to carefully apply the XP practice of "testing everything that could possibly break." Overzealous application can result in redundant testing of proven functionality. You'll notice this when you're working in different parts of the system and you see similar-looking unit tests. Another warning sign is if you or your partner are cutting and pasting parts of unit test code from one test class to another.

Fortunately, everyone on the team saw the value of unit tests, but I've also been on client sites where selling unit testing to developers was more challenging. An unobtrusive approach is to start by convincing developers that writing a unit test for each bug they fix will guarantee that the bug will never be reintroduced at a later date. Another tactic is to first get developers to buy into refactoring and then remind them that you can't refactor effectively without unit tests.

Whatever approach you take, developers must buy into unit tests. Unit tests are the foundation on which much of XP rests. It lets developers make changes to the code base with confidence, minimizes the chances of reintroducing bugs into the system, and allows the team to refactor continuously and aggressively.

Test-driven development

TDD is a software development technique in which you write unit tests prior to implementation.⁵ Initially, most developers on the team were unfamiliar with the concept of writing test code before the implementation. Introducing TDD took time because of the number of skills needed before we could apply this practice with confidence. Developers must first be comfortable with the mechanics of writing unit tests. Strong object-oriented skills and a philosophy of simple design also aided in applying TDD.

Developers often view test-first design as only a testing technique. However, when applied effectively, test-first design also helps de-

velopers enhance their OO designs. Tests are an important side effect, but more important is the resulting design the technique helps produce. Pairing was critical when introducing TDD to the team. On our project, TDD tended to occur only when one or both developers in a pair were keen on its application. Working with someone confident and able to demonstrate TDD by example went a long way into bringing others on board. When people worked alone or with others unfamiliar with TDD, the pair often slipped back into their more comfortable nontest habits. Introducing the practice into new environments requires patience and persistence. Ultimately, however, approximately one-third of the code was implemented with TDD.

Refactoring

Refactoring changes a software system so that it doesn't alter the code's external behavior yet improves its internal structure.⁴ If simplicity is the destination, refactoring is the vehicle that gets us there. Taken in isolation, each refactoring is quite simple. How to refactor effectively is not so obvious. People new to refactoring are sometimes unclear about when to refactor, and to what extent. Management can view refactoring negatively when it's misapplied. I once witnessed a development team who delayed all refactoring and testing until the last three days of a three-week iteration. Over time, the code became more resistant to change, and introducing new functionality became nontrivial and time-consuming. When asked why new features took so long to implement, developers replied that they were having to spend large amounts of time refactoring the code. This resulted in missed deadlines, unhappy customers, and management who wanted to hear nothing more about this practice called refactoring. It's not fair to blame improperly applied refactoring as the sole cause of all this project's woes. However, I've seen teams make refactoring a scapegoat for missed deadlines and other project ills. Developers should avoid this kind of situation because refactoring plays an important role on XP projects and would be a beneficial practice regardless of the development methodology used.

One of XP's cornerstones is that the code must be in its cleanest and simplest state at all times,¹ letting teams more easily handle future requirement changes. This is easiest to accomplish when developers refactor continuously,

Pairing was critical when introducing TDD to the team. On our project, TDD tended to occur only when one or both developers in a pair were keen on its application.

Why overdesign a system in a vain attempt to anticipate future requirement changes? Instead, start simply and let the code tell you whether you need to use a pattern.

as opposed to setting aside dedicated time purely for refactoring.

When we did not refactor continuously, the code became more cumbersome and resistant to change. Once again, pairing those new to refactoring with those more experienced helped demonstrate this technique's effective application. When faced with large refactorings, use stand-up meetings to share your insights and design goals for that part of the application. Put warning signs in the code to prevent others from propagating the undesired design. Over time, you and the rest of the team will be able to move toward the simpler, cleaner solution while simultaneously delivering new functionality.

Simplest thing possible

Everyone on the team agreed that we wanted to always do the simplest thing possible when writing code. However, defining simplicity was challenging, particularly relative to the use of design patterns.⁶ Some team members were quite comfortable with multiple layers of indirection and applying design patterns. However, for others, the prescribed design pattern solutions often seemed overly complex for the problem at hand. The XP literature reminds us that we should always “do the simplest thing that could possibly work.” Any extra work we do anticipating future requirements is often unwarranted and misapplied. Prematurely applying design patterns is an example of unnecessary complexity in the code—for example, a Strategy pattern with only one strategy.

Design patterns and XP practices appear to clash on the issue of simplicity. Joshua Kerievsky presented the best explanation I've seen regarding this conundrum.⁷ Kerievsky points out that the XP simplicity practice has definite merit. Why overdesign a system in a vain attempt to anticipate future requirement changes? Instead, start simply and let the code tell you whether the use of a pattern is warranted. Design patterns should be targets for our refactorings. Refactor into the pattern rather than start there. This simple advice is powerful and yields a good balance between XP's quest for simplicity and patterns' expressive power.

Continuous integration

Where JUnit aided us in testing, another open source project called Ant (<http://jakarta.apache.org/ant/index.html>) helped us continuously integrate our code into the master build.

Ant build scripts resemble make files in Unix. Our Ant build scripts could check out our latest code, compile it, release it to our target environment, and run all unit tests through a single command. Because Ant is written in Java, we were able to run the same build process on various development environments.

We also used an open source project started at ThoughtWorks called CruiseControl (<http://cruisecontrol.sourceforge.net>). CruiseControl periodically scans version control systems for newly checked-in code. When new code is checked in, CruiseControl runs the build scripts and promptly reports if anything is wrong.

By continuously checking in code throughout the day and having a build process monitor the application's state, the team avoided much of the pain traditionally associated with software integration.

Looking back, I wish we'd put more emphasis on making the tests run faster, without compromising the ability to test the system exactly as it would run in production. At the project's end, our system took 24 minutes to build. As a result, developers started running a subset of the tests only before checking in code—less than ideal. One way to speed up our tests would be to use MockObjects to fake out Web container and database calls. Another option would be to direct our tests against fast, lightweight, in-memory databases instead of their full-blown production counterparts.

The environment

Like any ecosystem in nature, certain environmental conditions help its inhabitants live and thrive. This project's optimal ecosystem was when the team was colocated, working within earshot of one another. Big open spaces with large tables where developers could sit side by side and pair-program were most conducive to a productive work environment. The environment resembled the “caves and common” room layout described by Ken Auer.⁸ In this scenario, there is a common area for pairs and groups of developers to congregate and a caves area for people when they require privacy. This seems to strike a nice balance between the high-bandwidth pairing environment and the more secluded private space we all sometimes need.

Our project did experience a setback, however, regarding the environment. About three quarters through the project, the team was moved into another building. Consequently,

we lost our big open area and were placed into traditional small offices. The immediate effects were that team members suddenly spent a greater part of their day getting up and walking the halls searching for the people they needed. Communication between team members suffered. Stand-up meetings then played an even more central role in coordinating and organizing the team.

It's the little things that count

Like all things in life, the little things often end up making the biggest differences. Little pats on the back were greatly appreciated. Management realized that by making the developers feel genuinely appreciated, developers continuously went that extra mile. Tasty snacks for those long iteration-planning meetings and occasional mid-afternoon coffee runs helped keep the project fun and the morale high. The QA department (our tester) also demonstrated her appreciation to developers who wrote a thorough set of unit tests by rewarding them with stickers (much like the ones you might receive in grade school). These soon became sought-after items by developers. Although these small gestures might seem trivial by themselves, cumulatively they aided to the project's success. I encourage teams to look for ways of making their projects fun and to give each other occasional pats on the back for a job well done.

Release management

We found TCPL very receptive to how XP handles release management. We based our release cycles on two-week iterations, with a formal release in six months. Two weeks sufficed to predictably add new functionality to the system and to give management a sense of the project's progress. The two-week period also kept developers focused on delivering quickly, without being lulled into a sense of complacency that can occur with longer iteration periods. Story planning worked best when someone with authority, who could speak on behalf of the customer, was present. The client also appreciated the fact that you could reprioritize stories without hugely affecting the project schedule.

The XP metaphor

The XP *metaphor* is the words and terms used to describe the system's overall architecture. Before this project, I did not place the metaphor practice in high standing relative to

other XP practices such as unit testing or refactoring. Many of the experience reports and personal conversations with other teams practicing XP led me to believe that the metaphor wasn't important. My own experience showed this as we didn't talk about or raise the XP metaphor's profile at TCPL.

Upon reflection, however, I believe we did have a project metaphor for the project; we just didn't appreciate it at the time. We described the event notification system as having paging-like capabilities. Subscribers to business events could be either groups of people or single individuals. Furthermore, our system was built up by aggregating existing Web services to form a larger application. Although simplistic, these words and descriptions of what the system did played an important role in realizing it. The words used in the metaphor manifested themselves as abstractions in the software. That's the point of the metaphor.

Although those applying XP often overlook the metaphor, it's a very important aspect of the development process and shouldn't be overlooked. In future projects, I would aim to raise the metaphor's visibility and importance in the hopes of realizing better abstractions and insights in the code.

Room for improvement

Hindsight is 20/20. Given a second chance, we would do some things differently.

Communication with external groups

In the later development stages, we needed external services and expertise, such as database and system administrators for configuring and setting up our production environments. We set up stories within our iterations regarding interaction with these external groups. These stories became the most difficult and the longest to complete.

One mistake we made was assuming that these other groups would work within our time frames—two-week iteration periods. When working with external groups, it's helpful if you can send a scout ahead (preferably a project manager to talk to the external groups' leaders) to ensure everyone is on the same page and to minimize any roadblocks ahead of time. It might also have helped to invite external group members to a few of our stand-up meetings. Through their presence at these meetings, we could have gently shown them how we worked

Little pats on the back were greatly appreciated. Management realized that by making the developers feel genuinely appreciated, developers continuously went that extra mile.

It's hard to be neutral as a developer because of how XP practices can differ from more traditional forms of software development.

and shared with them our objectives and goals. More communication would have led to better task, activity, and schedule coordination. If you depend on an external group, start working with them sooner rather than later.

Get a real business customer

The project's direction was periodically challenged. Because the project's business backers periodically changed, so did the requirements. So, it wasn't always clear what business problem we were trying to solve. On one hand, our mandate was to replace a legacy application. On the other hand, the new technology we were working with could also help other groups within TCPL solve their business problems.

Story planning sometimes became tricky, and the analyst and project manager had to periodically juggle stories for multiple clients. Story planning would have been much easier if we had a single business customer continuously present on the project driving requirements.

Looking back, I believe we did a good job of gathering and implementing requirements from multiple stakeholders. How might things have turned out if we had a single strong business partner from the beginning? Would we have forgone certain technology investigative spikes? Might we have done things even more simply? Although not always feasible, I would strive for a single business customer.

Make testers a part of the team

We added our quality assurance resource, or tester, to the team partway through the project. As a developer who was continuously writing tests, it was not immediately clear to me what role this person would play. Would she write further tests we hadn't thought of? Perhaps she would devise a formal test plan verifying that the new application sufficiently matched the legacy application's functionality. Fortunately, our tester had the experience and foresight to see that a tester's role on an XP project differed from that of a QA-style project.

Initially, she primarily focused on the application's quality. She would pair test user stories with developers; she highlighted missed test cases and verified that a sufficient number of tests were being written. She collected metrics the team wanted to see (for instance, ratio of production code to test code and unit test coverage). She manually tested the application, looking at usability

issues with a focus on how well the program intent matched customer expectations.

As the project progressed, our tester's focus shifted. Although she always kept one eye on overall application quality, she began playing more of an analyst role on the team. She began attending story-planning meetings and helped flush out requirements for future iterations. She built trust with external customers by helping them write acceptance tests.

After reviewing this article, our tester commented that early XP literature did not describe what role testers should play on XP projects. It seemed that formal, traditional QA testers weren't part of the XP team. She asked me to answer the following question: "Did you find having a QA tester as part of the team invalidated or supplemented any of the other XP practices?" This question was easy to answer because our tester became a valuable team member and greatly contributed to the project's overall success. It's important to note, however, that she didn't play the traditional role of QA tester—that role didn't exist. Testing and quality are a team responsibility that you shouldn't relegate to an outside entity, such as a traditional QA group. Better results will be achieved when testers are made a core part of the team. They can aid customers in defining and automating acceptance tests as well as play the role of analyst and flush out hidden requirements and assumptions. Lisa Crispin and Tip House's book, *Testing Extreme Programming*, is a good reference for the role testers play on XP projects.⁹

How did we get people to do XP?

Pete McBreen points out that it's difficult to find software developers who are neutral on the subject of XP.¹⁰ It's hard to be neutral as a developer because of how XP practices can differ from more traditional forms of software development. For example, consider pair programming and TDD. Most developers today do not write code in pairs and do not write tests before implementing the solution in code. These practices do not come naturally for most developers, and many who tried applying XP on projects commented on the difficulty of getting developers to apply these two key practices.

So how did we manage to get most of the team applying XP's practices most of the time? Fortunately, on this project, most project participants were keen on applying XP by the book.

Table 1**Our experience with XP practices**

XP practice	Degree of adoption	Comments
Planning game	Full	The team met regularly for story planning. Stories were captured and tracked on an Excel spreadsheet.
Small releases	Full	Iterations took two weeks.
Metaphor	Partial	We developed models and words, not explicitly referred to as metaphor, to describe the system.
Simple design	Full	We kept the design simple, but we could have done things even more simply.
Test-driven development	Partial	Those comfortable with TDD wrote all code test first. Those developers unfamiliar with TDD slipped back into non-test-first periodically.
Refactoring	Full	We continuously refactored all code.
Pair programming	Full	Most code was worked on in pairs. This practice was crucial for demonstrating XP by example.
Collective ownership	Full	All developers could (and did) work with any area of the application at any time.
Continuous	Full	An automated build process compiled, tested, and deployed our software integration multiple times a day. This was done with CruiseControl, Ant, and JUnit.
Sustainable pace	Full	There was no overtime on the project. Developers typically worked about 40 hours a week.
On-site customer	Partial	The customer responsible for proving Web services technology was on-site. The customer of the legacy application we were replacing was not.
Coding standards	Full	The team discussed and agreed on coding standards at the project's start.

We didn't have to defend XP or its practices; we simply had to apply them to the best of our ability. Even with strong backing from most of the management team, however, it was still a challenge to get everyone to continuously apply the practices at a high level. XP takes discipline.

Having good communicators and team players in key positions on the project played an important part in transitioning the team to XP. Without a strong iteration manager, demonstrating iterative development would have been challenging. The iteration manager performed many tasks. He ensured that stories were written in a timely manner and conducted the bi-weekly iteration meetings. He collected estimates and tracked developer velocity. In short, he helped set the project's rhythm.

The build master also played an important role. Although not represented formally in XP literature, this role helped in setting up project build files and enabling continuous integration services. A strong technical lead and developers who worked well with others rounded out the team. Having a critical mass of people (about half the team) on the project who were familiar with XP helped demonstrate both the spirit of XP and its practices.

Extensive pair programming best demonstrated routine XP practices. The very practice that many found difficult to apply on XP projects was a tremendous learning and communication tool for our team. Not only was pair

programming instrumental in demonstrating XP practices, it also spread domain knowledge throughout the team. Pair programming and colocated team members kept everyone on the same page and prevented problems from falling through the cracks.

Demonstrating good citizenship toward other team members also helped create a team-oriented culture supportive of XP practices. For instance, it was not uncommon during stand-up meetings for developers to offer their help when other team members were having trouble with their stories. The team leaders reminded everyone that it was acceptable to stop and help others, sometimes at the expense of their own work. A culture built on acts such as this helped form a strong team. People sincerely wanted to help one another and make the project a success. A secondary effect of encouraging good citizenship was that the code was kept in a relatively clean state. Because developers felt that they had the time to do the right thing, many "broken windows"¹¹ were fixed before they became larger problems.

Finally, but perhaps most importantly, having team members who worked well with others helped gently introduce XP's more foreign practices such as pair programming and TDD. XP is a very social way of developing software. The success of its adoption is greatly enhanced if the people working together are effective

IEEE Software

How to Reach Us

Writers

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access <http://computer.org/software/author.htm>.

Letters to the Editor

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide your title, affiliation, and email address with your letter.

On the Web

Access <http://computer.org/software> for information about *IEEE Software*.

Subscribe

Visit <http://computer.org/subscribe>.

Subscription Change of Address

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

Missing or Damaged Copies

If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles

For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission

To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.

communicators, like working in teams, are humble, and enjoy learning from others. All of this is important to help create a “no blame” culture, where people aren’t afraid to help one another and do what is right for the project. Fortunately, the team members on this project possessed these characteristics.

This project was a success for all parties involved. We finished on time and on schedule. TCPL uses the system in production and is planning further enhancements. TCPL was very impressed with agile development methodologies and will continue to use them on future projects. As with any project, it helps tremendously if you have good people—good not only in their ability to work with technology and develop software but, more importantly, possessing good communication skills. XP is a team-oriented way of developing software and works best with people who can express their ideas and work well with others. Teams made up of good people will always find a way to make projects successful. ☺

Acknowledgments

This article would not have been possible without the help and support of many people. I thank Vivek Sharma, Dean Larsen, Jason Yip, Eric Liu, Paul Julius, David Fletcher, Olga Babkova, Janet Gregory, Brad Marlborough, Laurie Williams, and Tannis Rasmusson for their insightful comments on drafts of this article.

References

1. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.
2. L. Williams and R. Kessler, *Pair Programming Illuminated*, Addison-Wesley, 2002.
3. F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975.
4. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
5. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.
6. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
7. J. Kerievsky, “Patterns and XP,” *Extreme Programming Examined*, G. Succi and M. Marchesi, eds., Addison-Wesley, 2001.
8. K. Auer and R. Miller, *Extreme Programming Applied: Playing to Win*, Addison-Wesley, 2002.
9. L. Crispin and T. House, *Testing Extreme Programming*, Addison-Wesley, 2002.
10. P. McBreen, *Questioning Extreme Programming*, Addison-Wesley, 2003.
11. A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

About the Author



Jonathan Rasmusson is a computer engineer with ThoughtWorks. He enjoys building enterprise applications and searching for better ways to write software. His research interests include agile development methodologies and exploring the human side of software development. He received an MS in computer engineering from the University of Alberta, Canada. Contact him at ThoughtWorks, 805-10th Ave. SW, 3rd Floor, Calgary, Alberta, Canada T2R 0B4; jrasmusson@thoughtworks.com.