

# YP and Urban Simulation: Applying an Agile Programming Methodology in a Politically Tempestuous Domain

Bjorn Freeman-Benson and Alan Borning  
Department of Computer Science & Engineering  
University of Washington, Box 352350  
Seattle, Washington 98195  
{bnfb,borning}@cs.washington.edu

## Abstract

*YP is an agile programming methodology that has evolved over the past 15 years. Many of its features are common to other agile methodologies; its novel features include using a highly visible, physical software status indicator (a real traffic light), and a well-defined nested set of development cycles. It is also an exceptionally open process, with the current status of the development process visible to the customers, as well as the code and documentation. We are using YP in developing the software for UrbanSim, a sophisticated simulation system for modeling urban land use, transportation, and environmental impacts over periods of 20 or more years under alternate possible scenarios. Our purpose in developing UrbanSim is to support public deliberation and debate on such issues as building a new light rail system or freeway, or changing zoning or economic incentives, as well as on broader issues such as sustainable, livable cities, economic vitality, social equity, and environmental preservation. The domain of use is thus politically charged, with different stakeholders bringing strongly held values to the table. Our goal is to not favor particular stakeholder values in the simulation or its output, but rather to let different stakeholders evaluate the results in light of what is important to them. There are several implications of this for the development process. First, having credible, reliable code is important — and further, both the code itself and the development process that produced it should be open and inspectable, not a black box. Second, to allow us to respond quickly to different stakeholder values and concerns, a flexible agile development process is required.*

Preprint – to appear in  
*Proceedings of the  
2003 Agile Development Conference  
Salt Lake City, June 2003*

## 1 Introduction

*[On a ship at sea:] a tempestuous noise of  
thunder and lightning heard  
Enter a Ship-Master and a Boatswain*

*Master:* BOATSWAIN!

*Boatswain:* Here, master; what cheer?

*Master:* Good; speak to the mariners. Fall to 't, yarely<sup>1</sup>, or we run ourselves aground. Bestir, bestir. *Exit.*

*Enter Mariners*

*Boatswain:* Heigh, my hearts! cheerly, cheerly, my hearts! yare, yare! Take in the topsail. Tend to the master's whistle.—Blow till thou burst thy wind, if room enough!

William Shakespeare, *The Tempest*, Act I Scene I

Yare Programming (YP) is an agile programming methodology that has evolved over the past 15 years. It was originally developed in industrial settings, and is now being used in a major academic software development effort. Many of its features are common to other agile methodologies; some are unique. We describe some of the more interesting (such as the traffic light) in Section 2. Then, in Section 3, we describe our application domain, and in Section 4 the implications of working in such a politically tempestuous domain for the development process, including our adoption of exceptionally open process. Our development is done in Java; but the methodology is applicable to a variety of languages.

<sup>1</sup>yarely, *adv.* Quickly, promptly; nimbly, briskly. [10]

## 2 Features of the YP Methodology

### 2.1 Testing

*Prospero.* All thy vexations  
Were but my trials of thy love, and thou  
Hast strangely stood the test.  
Act IV Scene I

As in other agile development processes, we employ test-first development with an extensive battery of tests, including both unit tests and acceptance tests. The unit tests are written using JUnit ([www.junit.org](http://www.junit.org)) and Ward Cunningham's Framework for Integrated Tests (FIT, at <http://fit.c2.com/>). There are two dimensions of testing for the project: who writes the tests, and how often the tests are run.

The tests could be written by either the clients or by the developers. We have tried to encourage the clients to write tests as part of their feature specification process, but have had less success than we desire. We have incorporated tools such as FIT to make it easier for users to read and write tests, but have still not achieved the agile "holy grail" of having our clients write tests as executable specifications. We have had more success using Ward Cunningham's technique of sitting side-by-side with the clients to develop the acceptance tests. We ask gentle questions about what a good test might cover, thus prompting the client to think about and explain what the correct inputs and outputs for such a test would be. Then we do the actual typing to create the test using the FIT framework.

There is clearly a social barrier that we are approaching with our requests for acceptance tests from the clients. In each application of YP to date, the clients have seen the value of having the tests, and have been willing to help generate them, but they have not been willing to create them on their own. Developing better processes and technical support to help overcome this reluctance is an important area for future research.

The second dimension of testing is how often the tests are run. In a perfect world, tests would take negligible time, and thus all the tests could be run all the time. In the real world, some tests are quick and others are very time-consuming. Thus, YP divides the tests into two suites:

- The *Commit-Level Tests* are those run every time someone commits code to the repository. These are the tests considered "most likely to fail." Another future research area for YP is to incorporate an automated "most likely" determination system, such as the one developed by Microsoft Research [7].
- The *Nightly Tests* consider the other complicated cases that would be tedious and time consuming to run during the course of ordinary development. This category

includes such tests as the database driver tests, and others that aren't likely to be affected by small changes to the code base or test data. The complete set of all tests are run automatically each night.

YP follows XP by insisting on always having a working, releasable system. Developers work by repeatedly taking small, testable steps, rather than large ones. One of the benefits of an extensive, and automatically checked, regression suite is to ensure exactly that: a working system in which features that were working do not suddenly stop working. As with other agile processes, YP uses the process equivalent of programming-by-contract. The contract is that the test suite (equivalent to the assertions) succeed at the beginning and the end of each unit of work (equivalent to a method). Thus if the suite fails after a change, the developer can be sure that it was his or her modifications that caused the problem.

YP also follows XP by using Test First programming. The benefits of writing the tests first rather than second are well described elsewhere. We have experimented with both techniques in settings in which we have been using YP, and then measured the defect rates. The survey results strongly supported using Test First, and so we have adopted this methodology throughout our process.

### 2.2 Framework for Integrated Tests

*Caliban.* You taught me language; and my profit on 't  
Is, I know how to curse. The red plague rid you  
For learning me your language!  
Act I Scene II

Two of the problems we have encountered in creating extensive test suites are:

- Writing the test scaffolding and objects takes a significant amount of time; and
- Java code is not easy for non-programmers to read.

To mitigate these problems, we incorporated the Framework for Integrated Tests (FIT), as described above. This has allowed us to write *literate tests*. Literate tests are similar to literate programming in that they mix descriptive prose and executable tests. Both prose and tests are written in HTML and displayed with a standard browser. The tests are written as HTML tables: in our case, each row of the table defines an action to be performed, such as "use this input database" or "expect the following errors." Some of table rows contain nested tables that define input or output objects.

Because the FIT tests are HTML pages, the tests are easy for our customers to read and understand. Additionally, through the use of a WYSIWYG HTML editor and

the simple table structure, the FIT tests are easy to create and modify. Our input and output simulation objects are sets, arrays, dictionaries, and other very regular structures, so nested HTML tables have proven to be a much easier mechanism for describing them than Java constructors and methods in JUnit.

## 2.3 The Traffic Light

We display the results of our testing very visibly: using a physical traffic light (a used one from a transportation department surplus program<sup>2</sup>). Actually there are several: one in the hallway of our laboratory, two more in developer offices, and one virtual traffic light on the web. We use four color combinations: green when the build and all the tests have succeed, yellow when the build and tests are in progress, yellow and green together when the build has passed the point where it is likely to fail (in practice, this means that all the tests have passed and that the final installer and distribution are being produced), and red if any part of the build or tests has failed. The lights are controlled by our Tinderbox-like continuous build script (named Fireman)<sup>3</sup>, which sends TCP packets to selected PCs. Each of physical traffic lights is connected to one of the PCs using a Weeder Technologies ([www.weedtech.com](http://www.weedtech.com)) WTSSR board attached to the serial port. Each PC runs a simple server (written in Visual Basic) that waits for packets (e.g., “+yellow”) and then sends serial control information (e.g., “AW00100”) to the WTSSR board. Because the physical traffic lights are controlled via TCP, we can easily support a geographically distributed team.

The traffic light is a powerful symbol of the current state of the software. The web version at [www.urbansim.org/fireman](http://www.urbansim.org/fireman) makes the status visible to anyone else who might be interested (including you, Gentle Reader, if you wish), and in particular for our customers, although in a less compelling way than the physical device. The physical traffic light has a number of advantages over the web version or other status indicators:

- It is a push technology, and thus the stakeholders don't have to remember to check the status - it just *there*.
- It uses culturally familiar set of colors, and communicates with the stakeholders on both a conscious and a subconscious level, reassuring them that all is well (green), and strongly suggesting a repair action when all is not (red).

<sup>2</sup>We purchased our traffic lights from Scott Signal Company ([www.trafficlights.com/scottsig.htm](http://www.trafficlights.com/scottsig.htm)).

<sup>3</sup>There are many equivalent continuous build systems; we are currently switching from our internal tool (Fireman) to the open source Cruise Control system <http://cruisecontrol.sourceforge.net/>.

- Seeing, and due to our implementation using relays, hearing, frequent cycles in a day gives a good feeling of progress.

We have used the traffic light as part of applying YP in four other development projects as well. Interestingly, only one team in addition to ours is still using the light — the other teams disconnected their lights because they didn't like them being red so much of the time. Rather than fixing the underlying problem (that their system was sufficiently unstable that their regression tests would not pass reliably), they chose to “eliminate the messenger.” Only one of the teams acknowledged this decision explicitly. We plan to do a user study across multiple industrial and academic projects to investigate this interesting, and apparently counter-productive, behavior.

YP is by no means the first process to use a physical project status indicator, but *is* probably the first to use a traffic light for this purpose. In conversations with other team leads and software managers, we learned that a number of them had tried “failure indicators,” such as sirens, flashing lights, red lights, and so forth, and that each had cancelled the experiment after a few days. Apparently the use of negative reinforcement (a red light) without the corresponding positive reinforcement (a green light) was too damaging to morale. Our experience with the traffic light is quite the opposite: everyone who joins a YP team immediately reports a sense of comfort at the large (8-12”) green light glowing its message of “all is well with the build.” Or on the rare occasions when the software has failed the nightly build tests, as the staff arrives in the morning, in the winter's gloom, with the lab hallway illuminated by the red glow of the traffic light, it's clear that a) something has gone wrong, and b) it should get fixed as soon as possible. Another possible difference between our use of the traffic light and other projects' uses of indicators is that we acknowledge that mistakes happen and thus do not penalize a (repaired) red light. This approach differs from the standard “build breakage penalty” (donuts, dunce caps, etc.) that many development organization utilize.

## 2.4 Iteration Planning and the Dashboard

*Prospero.* Thou shalt be as free  
As mountain winds; but then exactly do  
All points of my command.

*Ariel.* To the syllable.  
Act I Scene II

As is common with agile processes, the YP planning process uses nested iterations:

- Long-Term Cycle (Multi-Year)
- Release Cycle (Three Month)
- Iteration Cycle (Two Week)
- Daily Cycle (One Day)
- Task Cycle (Few Hours)

Each of the cycles has checklists, duties, and roles associated with it. For example, the Task Cycle includes “synchronize with the CVS repository, run the tests, and write a test-first test for the task.” The Daily Cycle includes “check the dashboard, verify any tasks that have been resolved, and start a new task.” We use two week iterations: long enough to provide stability for the developers to avoid thrashing, but also short enough to provide flexibility for the modelers to adjust the end point without wasting effort. As is common, we freeze the iteration task lists and specifications to prevent creeping features from distracting the developers.

We use Taskzilla, a locally-modified version of the Bugzilla defect-tracking system ([www.bugzilla.org/](http://www.bugzilla.org/)), to keep track of work requests: not only bugs, but also enhancements, administrative requests, and other tasks. As with the traffic light, Taskzilla is visible to anyone who browses our website. Our customers, as well as members of our own staff, enter bugs and work requests into it. We then use a Wiki page with a table to record the tasks assigned to current and future iterations. Finally, we use a number of cron scripts to merge the Taskzilla and Wiki information together and update our project status “dashboard” web pages.

Originally, only the developers used Taskzilla, but now all the team members (including the modelers and HCI group) make use of it. This is of course useful for group coordination, but has also been important in fostering our value of cooperation (see Section 4).

The dashboard has been used in industrial settings as well, as part of applying YP there. Our experience with it has been similar to our traffic light experience: three-quarters of the companies that had eagerly adopted a dashboard turned it off after a few months because they did not want the status of the project to be that visible.

One of the missing bits in our tool set is that our work request system (Taskzilla) does not have a hierarchical request structure to match our nested iterations. Thus, while we have a good tracking system for Iterations, we do not have an equivalent one for Releases or other longer-term planning. We believe that this lack of visibility into the tracking is a direct cause of the relative de-emphasis of long-term planning in each of five YP instances to date.

## 2.5 Student Involvement and Turnover

*Prospero.* Our revels now are ended. These our actors,  
As I foretold you, were all spirits, and  
Are melted into air, into thin air . . .

Act IV Scene I

The development team includes two professional software engineers, plus (currently) three undergraduate majors in Computer Science & Engineering. The traditional XP methodology assumes that all developers are more or less equal. In a university environment, however, it is important to involve undergraduates in research, and while they are bright, enthusiastic, and hard-working, they don’t have the depth of experience of the senior developers. Our development processes accommodate this well — we have had excellent success with undergraduates working on the project, and the experience has been beneficial both for the project and for the undergraduates and their education.

Another attribute of undergraduates is, of course, that they come and go, both on a daily basis (they have classes to attend) and on a yearly basis (they graduate). However, unlike in the play (“And, like this insubstantial pageant faded, Leave not a rack behind”) it is essential that all team members participate in maintaining a solid, well-documented and tested code base that others may build on, so that when an undergraduate graduates and moves on, his or her contribution remains. We handle this bright, inexperienced, and transient workforce with a more traditional Chief Programmer or Animal Farm<sup>4</sup> team organization rather than the XP pair programming organization. The two staff members provide mentoring and design guidance to the rest of the team, as well as periods of pairing, but the intermittent schedules do not permit full-time pairing. We have compensated by including mandatory pre-checkin design and code reviews, wherein an experienced developer is given a tour of the design, documentation, tests, and code. Because the review is done just before checkin, all the tests should already have passed. The reviewer uses a checklist, and they pair-browse the code base using the Eclipse comparison tool.

We have made some informal comparisons between the review-based technique and full-time pairing (using both our own prior experience with full-time pairing, as well as that of colleagues). Unfortunately, these comparisons indicate that our reviews do not ensure the same architectural integrity, code knowledge and quality as full-time pairing. We believe some of the reasons for this lower quality include the lack of time (less than an hour a day versus eight hours a day), the context changes (the reviewer is typically working on his or her own task as well), and the lack of ownership (the reviewer does not have the same emotional investment in getting it right). However, in spite of these

<sup>4</sup>All developers are equal, but some are more equal than others.

shortcomings, the reviews seem to be a reasonable compromise given the constraints of our academic setting.

## 2.6 Refactoring

*Trinculo.* What have we here? A man or a fish?  
Dead or alive? A fish; he smells like a fish; a very  
ancient and fish-like smell . . .  
Act II Scene II

As in other agile methodologies, an important aspect of our development work is nearly-continuous refactoring of the code, to maintain quality and to avoid ancient and fish-like smells of any kind.<sup>5</sup> This refactoring is, of course, integrated with our testing methodology and with the philosophy of taking small steps.

We end up doing more refactoring than other projects of comparable size, due to our staff attributes and turnover. However, because we consider refactoring to be a first-class citizen in the planning process, the long-term effect has been to keep the code comprehensible by non-programmers. One of the important goals of the UrbanSim project is to provide a transparent system (see below), and thus providing not only a valid simulation, but also the correct set of abstractions that are understandable by the customers.

## 2.7 Status Meetings and Today Email

YP, like XP, eschews status meetings. Instead, we report and plan with two different mechanisms. First, the daily “Stand Up” is a time to briefly discuss what one is planning to do that day, and to coordinate with others on those tasks. The stand up meeting is not used for status: it is a forward looking meeting. (It is a stand-up meeting to keep it short — if someone is leaning on the wall or sitting down, this indicates the meeting is going on too long.) Second, at the end of each day, a daily ‘today’ email is used for status. Each person sends a brief email to the entire group describing what they accomplished that day. The idea is that group status is essential to know, but boring to listen to. Since reading is faster than listening, we use short descriptive email status messages rather than status meetings. Additionally, the ‘today’ emails are low overhead and can be archived and reviewed, unlike the discussions in status meetings.

In our team, ‘today’ messages were initially used just by the developers, but have now spread to all project participants. In addition to their use as a status report, they had the unexpected additional effect of increasing group task

<sup>5</sup>N.B.: unfortunately, this does not extend to our current physical space. (Note to self: remember to speak to landlord about strange odor in back room.)

awareness at very low cost. We discuss this phenomenon (including survey results) in a short paper [2].

## 2.8 Making It All Work

*Sebastian.* Look, he’s winding up the watch of  
his wit; by and by it will strike.  
Act II Scene I

One aspect of software development that YP does not solve nor, as far as we can tell, does any other agile method solve, is that it’s still work. Keeping the process live and healthy requires continual diligence. Just because YP is an agile process does not mean that it’s easy, nor a magic bullet for software development. We still require a strong process leader and/or coach to keep the team on track and to evolve the process to meet the project’s changing needs.

## 3 UrbanSim

Patterns of land use and available transportation systems play a critical role in determining the economic vitality, livability, and sustainability of urban areas. Transportation interacts strongly with land use. For example, automobile-oriented development induces demand for more roads and parking (which in turn induces more automobile-oriented development), while compact, pedestrian-friendly urban environments can induce more walking and demand for transit. Both land use and transportation have strong environmental effects, in particular on emissions, resource consumption, and conversion of rural to suburban or urban land.

The process of planning and constructing a new light rail system or freeway, setting an urban growth boundary, changing tax policy, or modifying zoning and land use plans is often politically charged. Strong technical support can play a critical role in fostering informed civic deliberation and debate on these issues, as well as on broader issues such as sustainable, livable cities, economic vitality, social equity, and environmental preservation. We want urban planners and stakeholders to be able to consider different scenarios — packages of possible policies and investments — and then, based on these alternatives, model the resulting patterns of urban growth and redevelopment, of transportation usage, and of resource consumption and other environmental impacts, over periods of twenty or more years.

### 3.1 Technical Characteristics

UrbanSim [11, 12] is a simulation system that is intended to provide such technical support. It performs simulations of urban development, including transportation, land use, environmental impacts, and their interactions. It is a moderate-sized Java program (around 60,000 lines of code),

and is distributed as Open Source software (available from [www.urbansim.org](http://www.urbansim.org)). UrbanSim consists of a set of interacting component models that simulate different actors or processes within the urban environment. For example, the Residential Location Choice model simulates the process of household location — of a household deciding whether to rent or buy a dwelling, what kind (detached house, townhouse, apartment, etc.), and in what part of the city. This choice is modeled using a logit model. This is a probabilistic decision tree, in which the probability of a given choice is determined by both household characteristics (number of workers, number of children, household income, etc.), and the characteristics of a potential site. These probabilities are calibrated to observed data. Another model is the Developer Model, which simulates the activities of real estate developers (not software developers!) as they decide whether to develop new housing or commercial space, or redevelop existing space.

Although the total code size is moderate, many of the component models are complex conceptually and mathematically, drawing on results from urban economics, sociology, civil engineering, and other disciplines. The system is still undergoing considerable development, with the developers working in close collaboration with the domain experts. To support rapid software evolution and development, these component models are written as independently as possible, and communicate via a shared database rather than by direct method invocation [9].

### 3.2 Applications

To date, UrbanSim has been applied in the metropolitan regions that include Eugene/Springfield, Oregon; Honolulu, Hawaii; Houston, Texas; and Salt Lake City, Utah; application to the Puget Sound region (which includes Seattle, Washington) is under way. As part of the process of validating the model, we also performed a historical validation with the Eugene data, starting UrbanSim with the 1980 data, simulating until 1994, and then comparing with what actually happened.

In the United States, local or regional government planning agencies are typically charged with performing travel demand forecasting and land use modeling and forecasting. Each metropolitan region over a given size is required to have an identified “Metropolitan Planning Organization” (MPO) to receive federal funding. For example, in the Seattle area the MPO is the Puget Sound Regional Council; in the Salt Lake City area it is the Wasatch Front Regional Council. These MPOs are our primary customers.

UrbanSim has been recently brought into the middle of a land use and transportation dispute in Salt Lake City. A new freeway had been planned for the region, and after years of controversy, construction was imminent. In

response, two environmental groups (the Sierra Club and Utahns for Better Transportation), joined by the mayor of Salt Lake City, brought a lawsuit, wherein they claimed that the potential land use and environmental impacts of the proposed freeway had not been adequately evaluated, as required by law. In a June 2002 out-of-court settlement, all parties agreed to test the application of UrbanSim in the region, so that it could be used, for example, to model building or not building projects such as the freeway. Given pressures of this kind, reliable, high-quality, credible software is essential.

### 3.3 Using Value Sensitive Design

Clearly, the application domain is politically charged. Different stakeholders, such as business owners, members of advocacy or neighborhoods, elected officials, and planners, as well as other citizens of the region, may bring to the table widely divergent values about land use, transportation, and environmental impacts. To handle these issues in a comprehensive way, we are applying the emerging technique of Value Sensitive Design [5]. Value Sensitive Design is a theoretically grounded approach to the design of technology that accounts for human values in a principled and comprehensive manner throughout the design process.

For UrbanSim, we distinguish between *explicitly supported values* (i.e., ones that we explicitly want to embed in the simulation) and stakeholder values (i.e., ones that are important to some but not necessarily all of the stakeholders). Three explicitly supported values to which we have committed are fairness, accountability, and democracy. Examples of stakeholder values are environmental sustainability, walkable neighborhoods, space for business expansion, affordable housing, freight mobility, minimal government intervention, minimal commute time, open space preservation, property rights, and environmental justice. In contrast to the explicitly supported values, these stakeholder values may often be in conflict.

In more detail, one explicitly supported value is fairness, and more specifically freedom from bias. The simulation should not discriminate unfairly against any group of stakeholders. A second is accountability. Insofar as possible, stakeholders should be able to confirm that their values are reflected in the simulation, evaluate and judge its validity, and develop an appropriate level of confidence in its output. The third is democracy. The simulation should support the democratic process in the context of land use, transportation, and environmental planning. In turn, as part of supporting the democratic process, we decided that the model should not a priori rule out any one set of stakeholder values, but instead, should allow different stakeholders to evaluate the alternatives according to the values that are important to them.

A major issue is how to present the results of the simulation in useful ways for different stakeholders, particularly in light of widely divergent values about land use, transportation, and environmental impacts. We are relying heavily on the use of *indicators* for this purpose: numeric quantities that concisely distill attributes of concern about a situation. For example, for stakeholders interested in open space, an obvious indicator is the percentage of open space in a given area. For stakeholders concerned about environmental impacts, an indicator of air quality could be the number of days per year that air quality falls below EPA minimum requirements. For stakeholders concerned about freight mobility, an appropriate indicator might be the number of minutes of congestion delay per year per ton of freight. Other indicators are useful in evaluating the effectiveness of policies. For example, a pair of indicators to help evaluate the effectiveness of an urban growth boundary would be the population growth inside and outside the boundary.

Many of the technical choices in the design of the UrbanSim software are in response to the need to generate indicators and other evaluation measures that respond to different stakeholder values. For example, for some stakeholders, walkable, pedestrian-friendly neighborhoods are very important. But being able to model walking as a transportation mode makes difficult demands on the underlying simulation, requiring a finer-grained spatial scale than is needed for modeling automobile transportation alone. In turn, being able to answer questions about walking as a transportation mode supports two explicitly supported values: fairness (not to privilege one transportation mode over another), and democracy (since it is an important value to a significant number of stakeholders).

## 4 Implications for the Software Development Process

Given the high stakes and pressures of the kind described in Section 3.2, reliable, high-quality, credible software is essential. To help establish credibility, *repeatable* high quality is also essential.

UrbanSim's software architecture is designed to support rapid evolution in response to changed or additional requirements. For instance, the component models are designed to be easily reconfigured. Also, the system writes the simulation results into an SQL database, so that we can conveniently query it to produce new indicators quickly and as needed, as opposed to embedding the indicator code directly in the component models. Our YP development process is similarly tuned to agility and flexibility in the face of changing requirements.

We identify a set of important values for the development process:

**openness and accountability** This is in support of our explicitly supported value of accountability (Section 3.3), and the need for credibility of the software. To enhance the openness and accountability of our development process, we use technical artifacts described in Section 2, such the Taskzilla task management system, the traffic light, the dashboard, and the automated testing regime. We also make these visible via the web to our customers as well as to our development team — for example, Taskzilla, the web version of the traffic light, and the dashboard are all available from our web page.

**collective ownership** This is important for at least two reasons. First, collective ownership mitigates the difficulty of developer turnover (particularly among students). Second, it increases reliability and quality, by bringing more eyes on each part of the code.

**cooperation** This is essential in creating and maintaining a productive, satisfying work environment. Cooperation is primarily fostered by the actions and attitudes of the members of the team, but in some cases changes to the procedures and technical artifacts help, such as including tasks for all of the team in Taskzilla and in iteration cycle planning, not just those of the developers. (This has helped foster cooperation, since now, for example, a problem can be entered as a bug in Taskzilla and then scheduled, without worrying whether it was due to a specification bug, an implementation bug, or some combination: it's just something that the team needs to fix.)

### 4.1 Role of Open Source

*Prospero.* Of temporal royalties  
He thinks me now incapable ...  
Act I Scene II

The UrbanSim software is all Open Source, licensed under the GNU General Public License [4]. Open Source licensing has significant benefits for software reliability, robustness, and support for sharing and collaboration, and is of course well-understood in the software development community. However, it is not yet a common model for land use or transportation modeling software — but we believe that it is quite appropriate for this domain, in which the development effort is publicly funded, the customers are government agencies, and the requirements for openness and credibility are so strong. Our intent is that allowing and encouraging access to the model without proprietary restrictions will stimulate rapid innovation and sharing among agencies.

As described in Section 3.2, the immediate clients for UrbanSim and systems like it are typically Metropolitan

Planning Organizations. These MPO's often contract with consulting companies to write or customize modeling software. Launching an operational land use and transportation model is a complex undertaking, Open Source software or not, and will involve considerable work in preparing the data and customizing the model to local conditions. This means that an Open Source platform, such as UrbanSim, can still provide a solid basis for a service-oriented business model for a consultancy.

Open Source doesn't just mean that the code is available, but it also implies that others can contribute to the project in a number of well defined roles: Users, Developers, Committers, and Architects (as defined in [www.eclipse.org/eclipse/eclipse-charter.html](http://www.eclipse.org/eclipse/eclipse-charter.html)); another role for our project is Modeler. (Also see e.g. [www.mozilla.org/about/roles.html](http://www.mozilla.org/about/roles.html) and [www.opensource.org/](http://www.opensource.org/).)

## 4.2 Beyond Open Source

YP is nicely suited to the "All Open" Development style of the UrbanSim project. By "All Open," we mean not just Open Source, but Open Everything:

**Open Source** means that availability and access to the source code, as well as the distribution rights (free redistribution, integrity of the author's source code, etc.)<sup>6</sup> are not compromised.

**Open Code** means that the source code is readable, documented, and understandable. Not only is the source code *not* obfuscated (an Open Source requirement), but that the source code is deliberately written to be as understandable by as many people as possible. (We are working on a complex and difficult problem, so there are obviously limits here, but we try to achieve this goal as well as we can.) Open Code also means that this principle of clarity is not compromised.

**Open Design** means that the design documents are available and easily accessed. Further, the design is deliberately created to be clear to as large an audience as possible.

**Open Process** means that the process documents and status are also available and easily accessed, and are (again) open, inspectable, and written for clarity. Open Process implies that such access cannot be compromised.

YP has a number of attributes that make it an Open Process:

- The status of the project is open and available for all to see via the project dashboard and the traffic light.
- The future directions of the project are open and available via the iteration and release plans on the website, as well as the Taskzilla list of tasks and defects.
- The history of the project, including all the dirty laundry of schedule slips and incorrect design decisions, is open and available via the website and Taskzilla. We are planning to create tools and webpages that will make it easier to examine the project history and plans, but all the information is available even without these tools.

Additionally, the process itself is open to changes. For example, we originally had only the most current successful build available on the download site. Based on feedback from the MPOs, we changed the process to include both stable builds and current builds (see Section 4.3 that follows). Further input has indicated that we need to include three levels of builds: releases, stable builds, and current builds. Thus an Open Process means that not only is the current, future, and historical status available for view, but that the community can contribute to the planning and the process.

## 4.3 Social Difficulties With Constant Change

One of the advantages that XP, YP, and other agile development processes have is the constantly updated and tested current build. We provide our latest successful build via a download site and an "update" menu item in the installed product. In the context of UrbanSim, we felt that this would be a big benefit to our customers, since they could report a problem (for example, a small bug in the simulation algorithm) one day and download a patched version the next day.

However, we discovered that many organizations, including the Metropolitan Planning Organizations who are our primary customers, prefer fewer releases, and so we have provided that option. We hypothesize that this may be due to one or more of the following reasons. First, they might prefer a known to an unknown set of defects. Second, their internal process for evaluating a new release have a longer cycle time. Or third, they might have developed a distrust of software due to its general unreliability, and are unaware of the extensive testing done on each release. We plan to gather data on this issue through empirical investigation (semi-structured interviews and surveys), and plan to report the results in a future paper.

## 4.4 Efficiency Considerations

*Prospero.* Fetch us in fuel; and be quick; . . .  
Act I Scene II

<sup>6</sup>[www.opensource.org/docs/definition.php](http://www.opensource.org/docs/definition.php)

UrbanSim is a fine-grained simulation, with much more spatial detail than other land use models. This makes significant demands on our software, both for execution speed and memory usage. Despite these demands, we have been satisfied with our decision to develop in Java rather than C or C++, because of the gains in programmer productivity (in particular not having to do explicit memory management). However, some additional work has been required, which has added to the programmer burden and the complexity of the code; but the overall tradeoff has still supported our choice.

Execution speed has been a particular issue in running the logit computations for the more complex component models, in particular for Residential Location Choice and the Real Estate Developer models. To compensate, we make extensive use of caching to store intermediate logit computations and reuse them.

Memory usage has also been an issue. The fine-grained simulation creates very large numbers of small objects. In Java, each object has 10-20 bytes of overhead, making it expensive to use small objects. In a technique developed by Michael Noth, we have handled this issue by “exploding” the small objects into large parallel arrays. For example, rather than 10,000,000 grid cell objects, each with an  $x$  field, a  $y$  field, and so forth, we use an array with 10,000,000 floats to hold the  $x$  values, another array to hold the  $y$  values, and so forth. In our Version 1 implementation of Urbansim [9], this exploded object representation was all-too-visible to the programmer — we achieved a substantial reduction in memory usage, but at a cost in code readability and maintainability, and perhaps more importantly, decreased openness.

Note that exploded objects are not the same as the Flyweight pattern [6]: the Flyweight pattern is a way to reduce storage for the shared fields, whereas exploded objects are a way to reduce storage for the instance specific fields. For example, the Flyweight pattern would share the storage for the grid cell *width* and *height* values (common to all grid cells), but not the  $x$  and  $y$  values. Exploded objects do not share storage, but rather move the information to a location where it can be stored more efficiently.

In the current implementation, we have retained the exploded object representation, but have wrapped it in a much cleaner interface. The array implementation of exploded objects is no longer visible to the programmer. Instead, programmers use a specialized iterator object, which returns a real Java object for each invocation of `next()`. To avoid the overhead of large amounts of object creation and garbage collection, this object is reused for successive invocations of `next()`. Therefore, the programmer should not keep other references to the object — lest the object be altered unexpectedly by another call to `next()` — but in all other respects it can be used as any other Java object. This

technique has worked reasonably well.

As a spinoff project, for his Ph.D. dissertation Michael Noth is developing a Java language extension that supports such exploded objects with convenient syntax and runtime support, which will make it more straightforward to use this exploded object representation.

## 5 Conclusion

*Miranda.* Your tale, sir, would cure deafness.  
Act I Scene II

YP (“one step beyond XP”) is an agile programming methodology that has been slowly evolving over the past 15 years, first in several industrial settings and most recently in a large academic software project. It includes a number of novel features (for example, the traffic light and the today emails), and has proven effective in helping to manage a substantial and rapidly changing development project.

We believe that it represents an interesting and useful point in the agile process space. The choices of what to include in — and what to exclude from — YP were made deliberately to tailor the process to the particular environment in which YP is being used: a team with some permanent and some student members, developing software for a research system that nevertheless must meet stringent quality and reliability requirements, and that is being applied in politically charged environments. Many of YP practices are identical to XP practices: short iterations, continuous integration, test first programming, collective code ownership, writing documentation as part of development, and so on. Some of the YP practices differ from XP, such as code reviews rather than pair programming, and using a online bug database rather than cards on the corkboard.

In addition to other agile methodologies, YP is also related the method used in the M.A.D. project [3]. In that project, an interdisciplinary team from Aarhus University in Denmark collaborated with a large shipping company in developing a prototype for a global customer service system. This project was notable in its adoption of a highly agile programming methodology, and integrating this methodology with ethnography and Participatory Design [1, 8]. As in the UrbanSim project, the researchers comprised a multidisciplinary team, which was simultaneously concerned with software development methodology, object-oriented design, and value issues (in this case arising from Participatory Design and its tradition of workplace democracy).

YP has evolved in a number of interesting ways in the process of its transition from industry to the academic environment, and in its current application to a politically tempestuous domain. First, the process has moved toward increasing openness. In its present form, the current build, the status of the build, our task list, and our progress are

all publicly visible. The intent with this strong visibility is to foster credibility and confidence — critical issues for our domain. Second, the process has expanded. Originally, only the developers participated in it; now the modelers and customers work with the same nested iteration cycles and Taskzilla management system.

We hope that others will be able to apply and adapt parts of the YP methodology to their own domains.

## Acknowledgments

We would particularly like to thank each and every member of the UrbanSim research team for their work on the project. We would like to thank the anonymous referees and the anonymous David Socha for their helpful comments: you have helped make this a better paper. The Earl of Oxford provided apt quotations. (Or maybe it was Will himself; we are agnostic on this question.) This research has been funded in part by National Science Foundation Grants EIA-0090832 and EIA-0121326.

## References

- [1] G. Bjerknes, P. Ehn, and M. Kyng, editors. *Computers and Democracy: A Scandinavian Challenge*. Averbury, Aldershot, U.K., 1987.
- [2] A. B. Brush and A. Borning. ‘Today’ messages: Lightweight group awareness via email. In *CHI 2003 Extended Abstracts*, pages 920–921. ACM Press, Apr. 2003.
- [3] M. Christensen, A. Crabtree, C. Damm, K. Hansen, O. Madsen, P. Marqvardsen, P. Mogensen, E. Sandvad, L. Sloth, and M. Thomsen. The M.A.D. experience: Multiperspective application development in evolutionary prototyping. In *Proceedings of ECOOP’98*, pages 13–40, Brussels, Belgium, 1998. Springer-Verlag. LNCS Volume 1445.
- [4] Free Software Foundation. GNU general public license, version 2. Web page: <http://www.gnu.org/copyleft/gpl.html>, 1991.
- [5] B. Friedman, P. Kahn, and A. Borning. Value Sensitive Design: Theory and methods. Technical Report 02-12-01, Dept. of Computer Science & Engineering, University of Washington, Seattle, Washington, 2002. Available from [www.urbansim.org/papers](http://www.urbansim.org/papers).
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [7] W. H. Gates. Keynote speech: The future of programming in a world of web services. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002. Text of speech available at <http://www.microsoft.com/billgates/speeches/2002/11-08oopsla.asp>.
- [8] J. Greenbaum and M. Kyng, editors. *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.
- [9] M. Noth, A. Borning, and P. Waddell. An extensible, modular architecture for simulating urban development, transportation, and environmental impacts. *Computers, Environment and Urban Systems*, 27(2):181–203, Mar. 2003.
- [10] J. A. Simpson and E. S. C. Weiner, editors. *Oxford English Dictionary*. Clarendon Press, Oxford, second edition, 1989. *OED Online*. Oxford University Press. Accessed 25 Jan. 2003.
- [11] P. Waddell. UrbanSim: Modeling urban development for land use, transportation, and environmental planning. *Journal of the American Planning Association*, 68(3):297–314, Summer 2002.
- [12] P. Waddell, A. Borning, M. Noth, N. Freier, M. Becke, and G. Ulfarsson. Microsimulation of urban development and location choices: Design and implementation of UrbanSim. *Networks and Spatial Economics*, 3(1):43–67, 2003.