Code Development with `seamless`: Introduction

Paul Adamson
paul.adamson.01@gmail.com

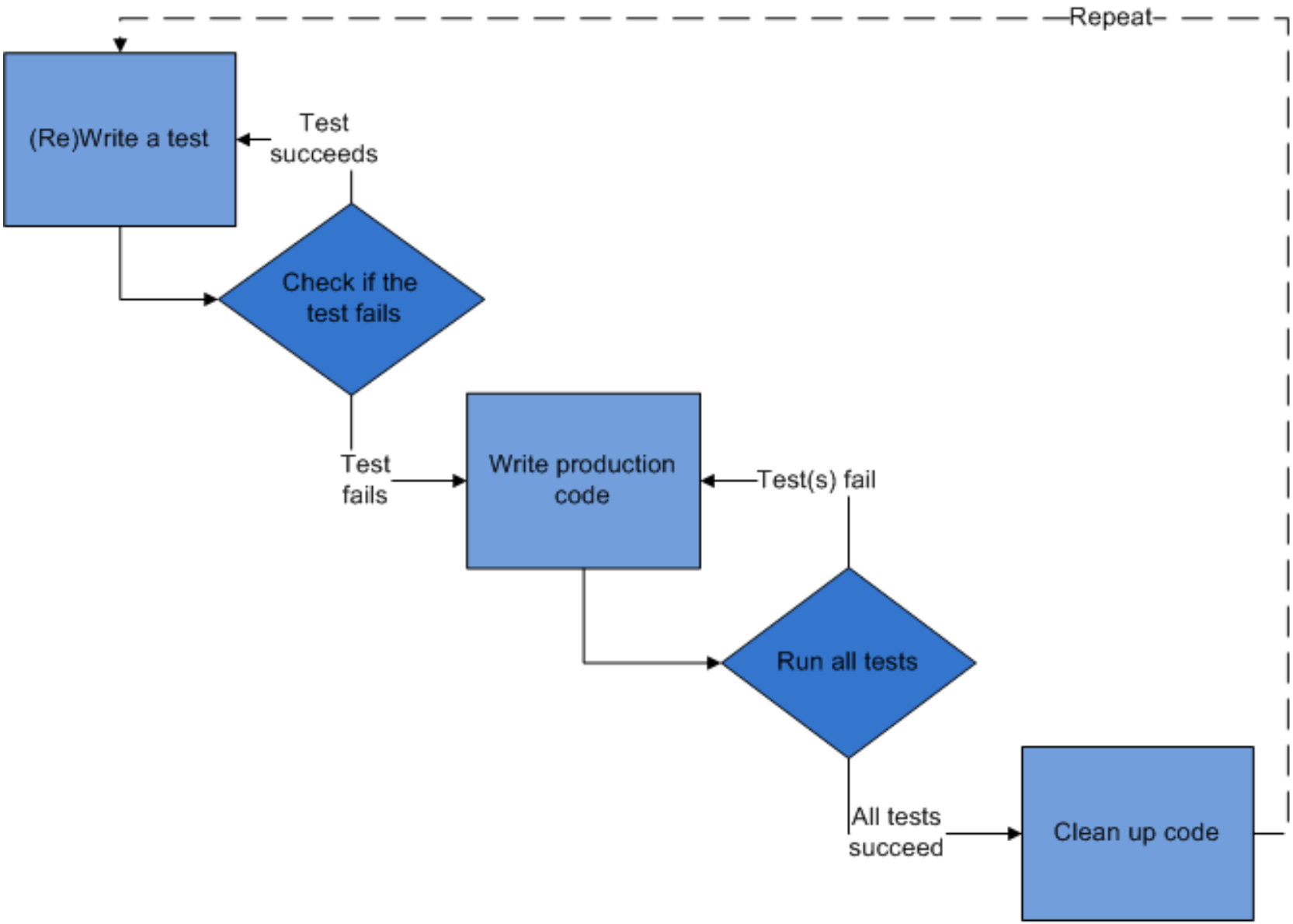
January 29, 2015

Outline

Motivation for `seamless`

- a lot of legacy code will be rewritten over the next couple of decades
- many incentives to producing reusable, maintainable code[3]
- some significant roadblocks
  - we are scientists first, programmers second (or third, fourth...)
  - high overhead of integrating multiple software engineering tools
  - we think *cost* $>$ *benefit*
- need a lightweight, easy to use system to implement good software engineering practices
- targeting small- to moderately-sized software projects
- not looking to develop a template for all projects—tool will be extensible & adaptable

Typical Test-Driven Development (TDD) Process[1]



Characteristics of Test-Driven Development

- *write the test* for a feature *before writing the code*
- aids design
  - forces systematic thinking about code functionality and interfaces
  - problem is decomposed into small, modular increments
- resulting code is robust and maintainable
  - completely covered by tests
  - accurate
  - low coupling, minimal side effects, high cohesion
  - well-factored, easier to modify
- tests are a form of documentation

Literate Programming

- reverse code/documentation weighting and development cycles
  - a lot of code with scant "comment delimited" plain text documentation versus
  - thorough, well-organized, and content-rich documentation with modular and efficient code
- high-level language code and associated documentation come from the same set of source files
- mathematics and graphics included in documentation

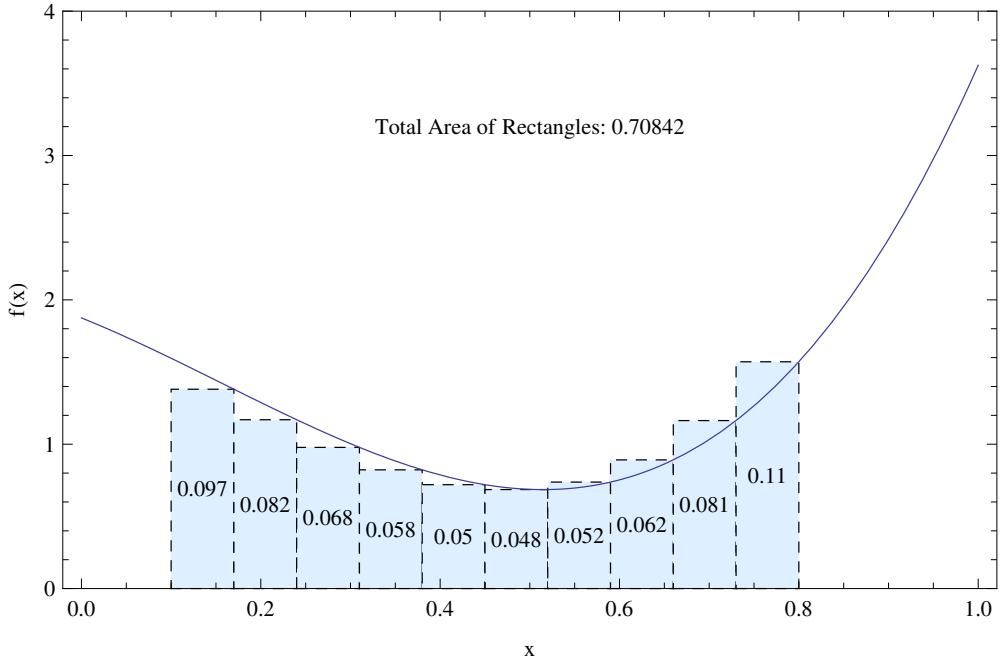
Literate Program Requirements[2]

1. code and documentation in same source
2. documentation and associated code adjacent
3. subdivided in a logical way
4. logical presentation versus conforming to syntactic constraints
5. includes open issues, rationales, etc.
6. description of the problem and solution (all the math and graphics necessary)
7. automatic cross references, indices, and different fonts for text, keywords, variable names, and literals
8. written in small chunks including documentation, definitions, and code

A Quasi-Literate Programming Example

We want to integrate the function
$f(x) = (2x - 0.5)^3 + (1.5x - 1)^2 - x + 1$ for x in $[0.1, 0.8]$

using the right rectangle numerical integration method. This method is illustrated in the figure to the right.



Helper (testFunctions.chpl). Provides the function
$f(x) = (2x - 0.5)^3 + (1.5x - 1)^2 - x + 1$.

```
proc f(x:real):real {
  return (2 * x - 0.5)**3 + (1.5 * x - 1)**2 - x + 1;
}
```

Un-`tangle` a Literate Program

- typical literate program presented in optimal order for readability
- `tangle` utility reorganizes code for compiler
- in example below, reference would point to error handling code
- compromise in `seamless`
  - follow TDD but borrow some goodness from literate programming
  - trade loose coupling and robust testing for code block referencing

```
proc readInData(filename:string) {
  var infile = open(filename, iomode.r);
  var reader = infile.reader();

  // 55 lines of error handling code

  readData(reader);
}
```

`seamless` Framework

- open source tools with a lightweight user interface

  user interface: text editor (*e.g.* vim) + make + LaTeX + git

  back end: python + compilers + LaTeX + cron
- requirements, documentation, specification, source code, and test suite all contained in a LaTeX document
- encourages good software engineering practices: traceability of requirements, robust testing, and modular design
- currently supports Chapel (`http://chapel.cray.com`), but extensible to other languages
- download: `http://www.github.com/padamson/seamless`

```
make // build PDF
make sources || tests // extract source or test code
make test // run tests
```

Documenting Good Requirements

- failing to write down good requirements is single biggest unnecessary risk a developer can take
- not understanding requirements diminishes productivity
- a thorough requirements specification is crucial for any non-trivial project (more than a few days of coding by one programmer)
- even trivial projects benefit from an informal specification
- tutorial gives example of scope and functional requirements
  - begin with the scope—a brief description of the software package, summarizing the code's high-level capabilities
  - functional requirements are documented in sufficient detail so that every line of code can be traced back to a requirement

Documenting Requirements in `seamless`

- requirements are nested in `description` environments
- each labeled item inherits the language of its higher level parents
- the most deeply nested items are labeled using the command `\req{x}`, where `x` is the desired number (*e.g.* `\req{1.1}`).

The code shall take inputs `a` and `b`

and `c`

R1.1 and compute `a + b - c`

R1.2 and compute `a + b + c`

R2 and compute `a * b`

- **R1.1:** the code shall take inputs `a` and `b` and `c` and compute `a + b - c`
- **R2:** the code shall take inputs `a` and `b` and compute `a * b`
- to reference a requirement, use the command `\ref{req@x}`, where `x` is the desired number (*e.g.* `\ref{req@1.1}`).

`seamless` Code Development Cycle

Once requirements are documented, the code development cycle follows:

1. Document a small part of the problem and its solution
    1. Include all aids at your disposal (*e.g.* math, graphics)
    2. Include references to requirements
2. Create a test (or tests, if appropriate)
    1. Keep the tests short
    2. Each test covers only one thing
    3. The test should run automatically
    4. Make sure the test fails
3. Create the code
    1. Document and label the code specification, referencing requirements as appropriate
    2. Write the simplest code possible to pass the test
    3. After the test passes, refactor to improve the code
    4. Run the tests again to ensure they still pass
    5. Refactor and retest some more
    6. Update the code specification if necessary

Characteristics of Resulting Code

- development cycle repeated until all requirements are met
- verify by checking the requirement traceability matrix
- process will help to ensure your code has the following characteristics:
  - completely documented
  - simple
  - readable
  - completely covered by tests
  - robust
  - accurate
  - maintainable
  - reusable

Tutorial Structure

The files in the provided tutorial can be adapted for your specific application:

Makefile supports pdflatex and latex to compile the LaTeX package into a PDF; also has targets to extract code and run the tests

seamless.cls provides the `seamless` document class (a modification of the "book" class)

seamless.sty provides a few LaTeX environments and commands

Numerical_Integration.tex the main LaTeX document with includes for the remaining LaTeX files

chapel_listing.tex used with the LaTeX `listings` package to properly format Chapel code

references.bib contains bibliography entries for use by bibtex

Tutorial Notation

Various environments are defined to highlight different types of notes within color-coded text boxes (*e.g.* `\begin{TODO}` and `\end{TODO}`):

TODO. Things that need to be done for the current version of the software. Inserted with `TODO` environment.

Note. Something of note that does not fit into any other category. Inserted with `note` environment.

Rationale. An explanation for a particular design choice. Inserted with `rationale` environment.

Tutorial Notation (continued)

Open issue. Issue that we do not know how to handle. Inserted with `openissue` environment.

Future. Issue or feature that we have a story about, but which is not yet fully-designed or implemented. Inserted with `future` environment.

In addition, the tutorial contains an additional type of text box to explain background information about the `seamless` framework:

`seamless`. Example text box used to provide background on the `seamless` approach in context of the tutorial. Inserted with `seamlessnote` environment.

Index

References

Test-driven development.
`http://en.wikipedia.org/wiki/Test-driven_development`.
Accessed: 2015-01-23.

Bart Childs.
Literate Programming, A Practitioner's View, pages 261–262.
Tugboat, December 1992.

Marian Petre and Greg Wilson.
Plos/mozilla scientific code review pilot: Summary of findings.
CoRR, abs/1311.2412, 2013.