Sabre Holdings™

Connecting people with the
world's greatest travel possibilities.

Test Driven Development

Java Developers Day

October 26, 2007

Agile Manifesto

- Agile Values:
  - ***Interactions with individuals** over processes and tools*.
    ***Creating working software** over comprehensive documentation.*
  - ***Customer collaboration** over contract negotiation*.
  - ***Responding to change** over following a plan.*

Fundamentally it is the application of common sense to methodology

Comes from the Agile Alliance www.agilemanifesto.org

Sabre Holdings

Practices

- **Planning**
  - User Stories
  - Customer Collaboration
  - Iteration Planning
  - Demo Sessions
  - Tracking Velocity
  - Release Planning
- **Design**
  - Simple design
  - YAGNI
  - Refactoring

- **Coding**
  - Coding Standards
  - Test Driven Development
  - Continious Integration
  - Common Code Ownership
  - Pair Programming
  - No overtime!
- **Quality Assurance**
  - Unit Testing
  - Integration Testing
  - Acceptance Testing
  - Testing Automization
  - Quality Metrics Tracking

Sabre Holdings

Test Driven Development

Traditional Waterfall Method:

Requirements ⇒ Design ⇒ **Code** ⇒ Test

Test Driven Development:

Requirements ⇒ **Test** ⇒ Design ⇒ Code

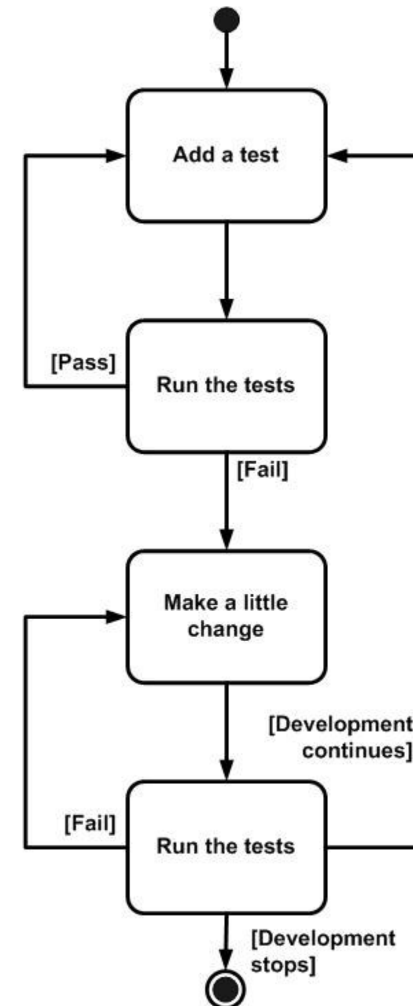
- **Write automated tests before you write code!**
- **Run the tests and prove they fail before you write code!**
- **If it is hard to write the test, then the design is poor!**

"Test First" enforces good OO design principles and enables refactoring**.**
It results in high quality code that is easy to change.

Sabre Holdings

It’s really simple!

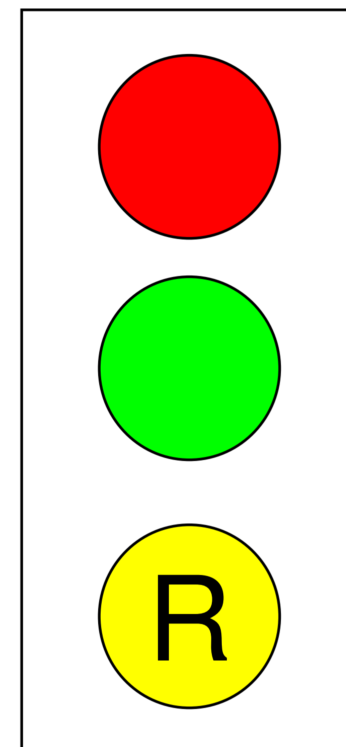
- Red - write a (failing) test
- Green - make the test working
- Refactor - ensure simplest design that passes the test



Copyright 2003 Scott W. Ambler

Keep the Rhythm!

- **Red - write a (failing) test**
  - Tell a story – invent the interface you wish you had
  - Include all of the elements in the story that you imagine are needed to get the right answers.
- **Green - make the test working**
  - Quick! Get the code to compile and get a green bar
  - Quick green excueses all sins, but only for a moment
- **Refactor - ensure simplest design that passes the test**
  - Remove the heinous sins you committed getting to green quickly
  - Remove duplication (both in the code and **tests)**
  - Get to green quickly

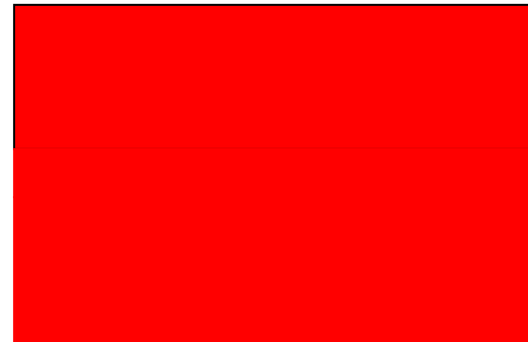


Sabre Holdings

TDD vs. Conventional Development

Test-Driven

Conventional



Programmer writes code for all functionality

Fixes compiler errors

Programmer develops functionality by alternating failing test, passing test

Functionality is proven and can be verified

Programmer thinks (hopes) it works

Debugs

Sabre Holdings

Requirements

- **Automated Tests**
  - Executed with single command
  - Single OK message if test passes
  - Covers **unit testing**, **acceptance tests** and **regression tests**
- **Built-in Tests**
  - Tests are part of application
  - Prepared by developer
  - Utility libraries – CppUnit, JUnit

Sabre Holdings

How to Begin?

- Start a ToDo list
- Start simply („start small or not at all”)
- Write automated test
- Write a little code
- Refactor to add design decisions one at a time

Sabre Holdings

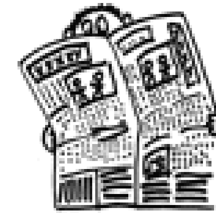
Test Types

- **Starter Test**
  - Where does the new operation belong?
- **Regression Tests**
  - If fixing a bug, add test that shows the bug
- **Learning Tests**
  - Use unit tests to learn API of third party libraries
- **Explanation Tests**
  - Communicate ideas using unit tests

Sabre Holdings

TDD Patterns

- **Isolated Tests**
  - Tests should be able to run in any order
- **Child Tests**
  - Break big test code to smaller tests



- **Mock Object**
  - Replace expensive or complicated resource with faked version
- **Self Shunt**
  - Use test class to test communication between objects
  - Allow other object to communicate with test class
- **Crash Test**
  - Test erroneous situations
- **Test Data**
  - Use data that are easy to read and follow
  - Test border conditions as well as typical data

Sabre Holdings

Self Shunt Test?

```
public class ScannerTest extends TestCase
{
  public void testScan () {
    Display display = new Display ();
    Scanner scanner = new Scanner (display);
    scanner.scan ();
  }
}
```

Sabre Holdings

Self Shunt Test ctd.

```
public class ScannerTest extends TestCase implements Display
{
  public void testScan () {
    // pass self as a display
    Scanner scanner = new Scanner (this);
    // scan calls displayItem on its display
    scanner.scan ();
    assertEquals (new Item ("Cornflakes"), this.lastItem);
  }

  // impl. of Display.displayItem ()
  void displayItem (Item item) {
    this.lastItem = item;
  }

  private Item lastItem;
}                    //http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf
```

Sabre Holdings

Unit Testing

- **Verify a class as a “standalone” unit**
  - Verify code logic
  - Verify its sending of/response to messages
    - Use mock objects to emulate collaborators, but only if necessary
    - What if understanding of messages is incorrect?
- **Verifies that we’ve constructed valid building blocks**
- **Feedback more rapid**

Sabre Holdings

Integration Testing

- **Interactions between modules in some (pseudo) deployed environment**
  - ATs are a form of integration testing *aka* system testing
- **Requires configuration, data, and environment setup**
  - Can we put this effort into ATs, which require the same?
- **Might provide more rapid feedback than ATs**
  - Does our understanding of the database still match reality?
  - Are we able to successfully configure some subset?

Sabre Holdings

Unit Tests and Integration Tests

- **Integration tests**
  - - Increased configuration/setup
  - - Slow feedback
  - + Test closer to “live” system
  - + Demonstrate adherence to requirements
- **Unit tests**
  - - Make assumptions about interactions
  - + Rapid feedback
  - + Usually easier to code
- **Either:**
  - Can miss things
    - Impossible to cover everything with tests. No silver bullets!
    - Agile allows us to learn and correct
  - Require effort to maintain
    - Mindset: things need to be documented.
    - Can we use the tests to help supplant written documents?
- **Well-balanced mixture of both essential**

Sabre Holdings™

What to test?

- **Any place you have fear**

Sabre Holdings™

What not to test

- **Getters/Setters**
- **toString()**
- **Standard Java API's (unless for learning purposes)**
- **Code from others (unless you distrust them)**
- **Classes introduced as part of refactoring (they're exercised when you get to green)**
- ***Private methods***

- **... unless you have fear**

Sabre Holdings

To-Do List

- What set of tests when passed, will demonstrate the presence of code we want?
- Make a list of the tests you know you need to have working

- Turn objections about the desugn into tests
- First, put on the list examples of every opeartion that you know you need to implement
- Next, for those operations that don’t already exist, put the null version of that operation on the list
- Finally list all of the refactoring that you think you will have to do in order to have clean code at the end of this session

- You may want a „now”, a „later” list (and possibly a „never”)
- **Don’t mark off a test until duplication has been removed**

Sabre Holdings

Code Coverage

- **It tells...**

  - what's not tested
  - not what's tested

Sabre Holdings

Test Smells

- Long setup code
- Setup duplication
- Long running tests
- Fragile tests
- Test data files

Sabre Holdings

Refactor in... ?

Code	– refactor in –	GREEN
Tests	– refactor in –	RED

Sabre Holdings

Challenges Faced with TDD

- **Databases**
  - Use Mocks or Fakes as “stand-ins” for the data access layer when testing higher-level objects
  - Fake Objects implement an interface but provide behavior that is controllable by the tests
  - Mock Objects add the ability to verify that implementation details and internal state are correct

Sabre Holdings ™

Challenges Faced with TDD

- **Sockets (and other badzewias)**
  - Use the actual socket for sending and receiving only (keep that layer thin)
  - Delegate message processing and preparation to other objects
  - Test the processing and preparation without using an actual socket

Sabre Holdings ™

Challenges Faced with TDD

- **Multi-threading**
  - Probably the toughest challenge I have faced
  - Test threads independently
  - Use timing and synchronization when testing thread interaction
  - Use internal state flags to ensure proper actions were executed

Sabre Holdings

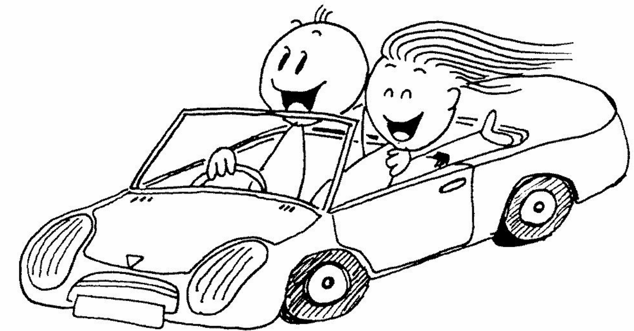
Challenges Faced with TDD

- **User Interfaces (UI)**
  - Keep the GUI layer thin
  - Create controller objects that handle logic and “hook them up” to UI events
  - See also
    - “The Humble Dialog” by Michael Feathers
    - Test-First User Interfaces (TFUI) at http://groups.yahoo.com/group/TestFirstUserInterfaces/

Sabre Holdings

Pair Programming

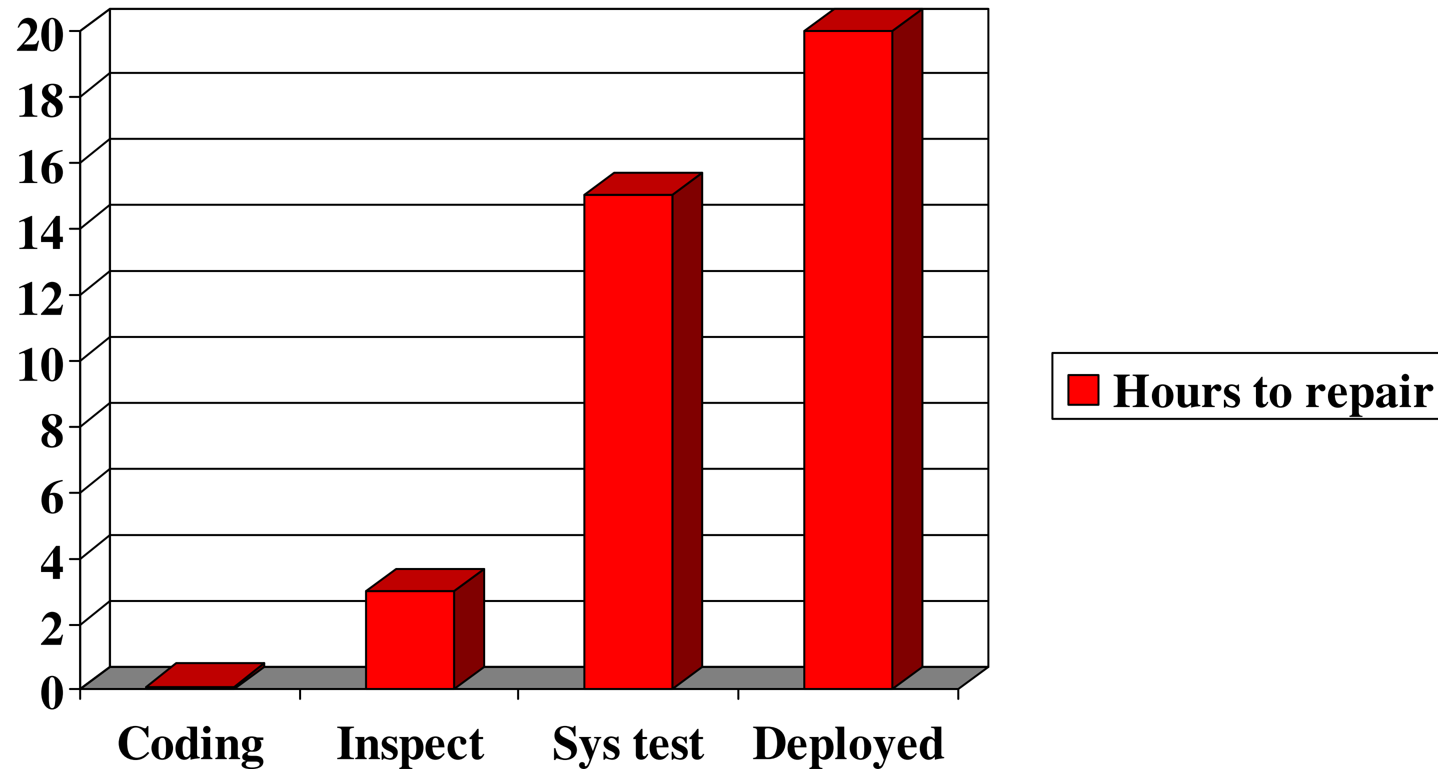
- Production software is best when built by two programmers, sitting side by side, at the same machine



- One member drives (tactical)
- One member navigates (strategic)
- Members rotate roles
- Pairs form and reform constantly within and between tasks
- Disagreement is healthy! Quiet pairs are dangerous.

Sabre Holdings

Pair Programming – Cost of DEFECTS



Cost of repairing a defect increases with the delay in detection

Sabre Holdings

Pair Programming

- Formal inspections rarely happen and then too late
- In Paired programming - inspections occur in real time - while cost of change is **lowest**.
- Reduces the '**truck count**' (lowered risk)
- Reduces **ramp up time** for new team members
- Easier to accommodate vacations and unplanned leave
- Increases overall team velocity
- Helps support the 'everybody is an architect' paradigm

Sabre Holdings ™

Dos

- Proper environment
- Division of roles: strategic and tactic
- Switch roles often
- Use as opportunity to learn everything
- Monitor pairs at the beginning
- Set ground rules for pairing
- Openly talk about successes and problems
- Expect (and accept) learning curve
- Switch pairs often

http://www.developer.com/lang/article.php/3652636

Sabre Holdings

Don’ts

- Force pairing on developers
- View pairing as one person watching, the other person doing
- Force the pairs or determine them ahead of time
- Sweat the impact of pairing on estimates
- Overdo it

- Be too overzealous about correct simple typos while in the strategic role
- Get too smart with "when you don't need to pair."
- Let one person dominate in a pairing session
- Let the pairings stagnate
- Give up on pairing without first giving it a fair trial

http://www.developer.com/lang/article.php/3652636

Simple Design

- **Rules of simple design**
  - Make it run
  - Make it right
  - Make it fast
- Ask ”**what is the simplest thing that could possibly work?"**
- All acceptance and unit tests must run (make it run)
- Express **everything** you need to express (make it right)
- No code is written without a test (make it right)
- Refactor Constantly! No duplication! (make it right)
- Use the fewest possible classes and methods (make it right)
- Adhere to or adapt the Metaphor (make it right)

Sabre Holdings

Simple Design

"*Design in XP is not a one-time thing, or an up-front thing, it is an all-the-time thing*" - Ron Jefferies

- Requirement changes are likely to supercede general solutions. Simple, tested designs are easy to change.

"*If you believe that the future is uncertain, and you believe that you can cheaply change your mind, then putting in functionality on speculation is crazy. Put in what you need when you need it.*" – Kent Beck

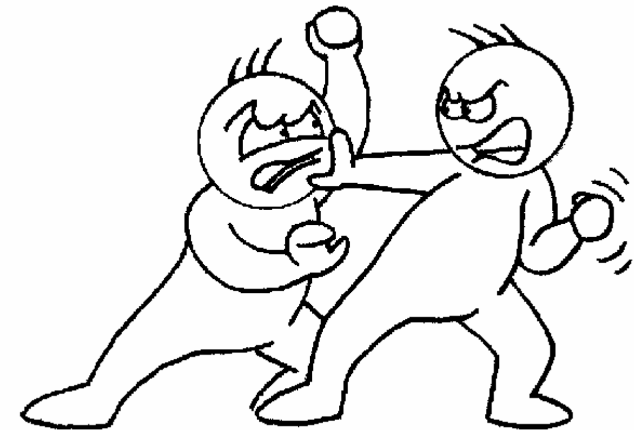
- Remember **YAGNI** (**You Aren't Going to Need It!**)
- Refactoring yields general solutions as they are required.
- Create the best design that can deliver the functionality today.

Sabre Holdings

Simple Design - vs. BDUF

"Draw all the diagrams you want but until you take a step you can't tell if it will actually work." - Kent Beck

- **BDUF** (Big Design Up Front) assumes that existing requirements are complete and won't change (or that you are prescient about future requirements)
- Simple Design assumes that existing requirements are incomplete and will change
- Simple Design delivers what is needed right now (lowers sunk cost)
- Quality is a function of systematic usage, not anticipatory design
- Simple design enables sustainable pace
- Never model for the sake of modeling
- Use UML as required to communicate concepts



Class Design Principles

- **Single Responsibility Principle**
  - Classes should have one reason to change
- **Open/Closed Principle**
  - Software modules should be open for extension,
  - closed for modification
- **Dependency Inversion Principle**
  - Details should depend on abstractions,
  - abstractions should not depend on details
- **Interface Segregation Principle**
  - Keep interfaces small
- **Liskov Substitution Principle**
  - Derived classes usable through base class interface,
  - without clients knowing the difference

Sabre Holdings ™

Continuous Integration

- Pairs check in code **constantly** (every 10 to 15 minutes)
- **100% of tests must run** before check in
- Automated build provide alerts if build breaks
- Must configure SCM for **unreserved check outs**.
- Enabled by automated testing
- Enables Collective Ownership
- Enables small Releases (daily build is the minimum)
- The last pair checking may have to merge manually
- You broke it? You fix it!
- A clean system can be shipped at any time

Sabre Holdings

Refactoring

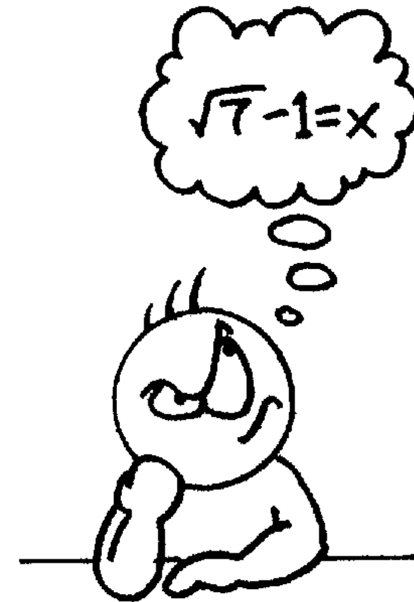
- "Refactoring is a disciplined technique for **restructuring an existing body of code**, altering its internal structure without changing its external behavior". - Martin Fowler

- Done in very **small steps** (don't debug: if a bug creeps in, undo!)
- **Done constantly** – like **water on a stone** - not postponed
  - Test -> Code -> Refactor -> Test -> Code -> Refactor -> etc.
- Supported by **automated unit tests** and automated (safe) refactoring tools.
- A good automated **refactoring tool** allows you to do basic refactorings even in the absence of unit tests.

Sabre Holdings™

Refactoring – Common Refactorings

- Extract Class
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass
- Introduce Parameter Object
- Move Method
- Rename Method
- Rename variable
- Replace Conditional with Polymorphism
- Replace Constructor with Factory Method
- Replace Magic Number with Symbolic Constant
- Replace Temp with Query



Sabre Holdings

Refactoring

"Any dummy can write code a computer can understand."

- **Since all code is documentation it must be readable.**

- Continuous **evolutionary design** depends on refactoring
- Enables **Simple Designs** to remain simple
- Refactoring creates **better** opportunities for design **optimizations**
- The worst smell is **duplicated code** (sometimes its hard to spot)
- Makes insights and discovery 'live' in the code

"Refactoring is not required if you are omniscient and perfect (and so is everyone else on the team!)"

Sabre Holdings

Practices Revisited

- **Planning**
  - User Stories
  - Customer Collaboration
  - Iteration Planning
  - Demo Sessions
  - Tracking Velocity
  - Release Planning
- **Design**
  - Simple design
  - YAGNI
  - Refactoring

- **Coding**
  - Coding Standards
  - Test Driven Development
  - Continious Integration
  - Common Code Ownership
  - Pair Programming
  - No overtime!
- **Quality Assurance**
  - Unit Testing
  - Integration Testing
  - Acceptance Testing
  - Testing Automization
  - Quality Metrics Tracking

Sabre Holdings

Test Driven Development

- Focus on **what is required**, not how to code it.
- Can't figure out what to test? - **communicate!**
- **Enables simple design** (safe guards against design for designs sake)
- Automated Acceptance tests are the **ultimate functional specification** - provide constant feedback and confidence
- Automated unit test cases are **insurance policies** - they alert team when somebody breaks the code.
- TDD practically **eliminates debugging**
- Full Regression tests are performed constantly
- Tests are upgraded if a defect *is* found - prevents a reoccurrence.
- Good unit tests tell you exactly what failed
- Unit tests serve as part of the system documentation

"You only have to test the stuff you'd like to work." -Ron Jefferies

Sabre Holdings

What *can't* be fully Test Driven?

- **Security Software**
- **Concurrency**
- **GUI**
- **Distributed Object Systems**
- **Database Schemas**
- **Language compiler/interpreter from BNF to production quality implementation**

- **However, unit tests in aforementioned areas are highly desired!**

Sabre Holdings

TDD Pros

- **Clean code that works**
  Better software quality, bugs get caught very early
- **Interface focused – well defined, usable interface**
- **System-level documentation**
- **Good design – high cohesion low coupling**
  Cheaper maintenance
- **Reusable code – code born with two clients**
  Code reuse savings
- **„Refactorable" code – refactoring results in equally tested code**
- **You know when you've finished**
- **Early defect recognition**
  Cheaper defect removal

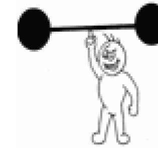
Sabre Holdings

Lessons Learned with TDD

- **TDD is hard**
  - It requires more discipline than conventional programming techniques
  - Success is dependent on having appropriate, accurate, and sufficient test cases

Sabre Holdings

Lessons Learned with TDD



- **TDD inspires confidence**
  - Tests prove the code works
  - “Safety Net” sensation encourages exploration and experimentation

Sabre Holdings

Lessons Learned with TDD

- **TDD improves productivity**
  - Problems are identified and corrected early in development
  - Less time is spent debugging
  - Tests provide a roadmap for development

Sabre Holdings

Lessons Learned with TDD

- **TDD leads to better software!**
  - Resulting code is simpler and easier to maintain
  - Tests provide on-going verification that the software works

Sabre Holdings

References

- http://www.testing.com
- http://www.mockobjects.com
- http://www.testdriven.com
- http://www.agiledata.org/essays/tdd.html
- http://www.objectmentor.com/writeUps/TestDrivenDevelopment

- **Jeff Langr/ Agile Java(TM): Crafting Code with Test-Driven Development (Robert C. Martin Series)**
- **Astels, David. *Test-Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall, 2003**
- **Beck, Kent. *Test-Driven Development: by Example.* Boston, MA: Addison-Wesley, 2003**
- **Jeffries, Ron (June 2004). *XProgramming.com: An Extreme Programming Resource.* [Online]. Available http://www.xprogramming.com**
- **Martin, Robert. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall, 2003**

Clip arts come from
http://tell.fll.purdue.edu/JapanProj/FLClipart/