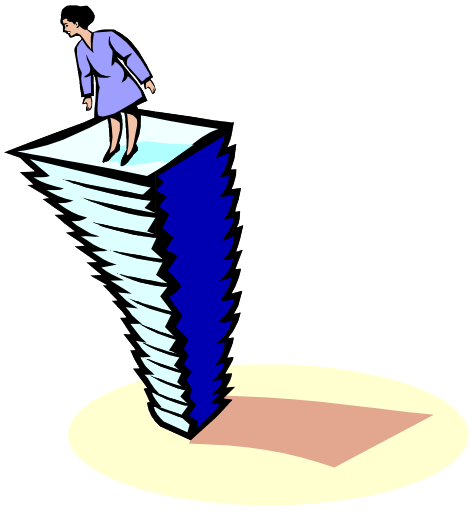
Beyond Green Field Software Development

Dennis Mancl, Dennis.Mancl@alcatel-lucent.com

Some basic concepts

Legacy Software

- Any software that **still provides you some value**
  - It might be **the only way** to collect and process certain kinds of data
  - The software is often the best documentation of your "business process"
  - It might be **ugly**, written in an **old programming language**, and it might be really **hard to modify**



Green field development

- You are solving a new problem, or you don't need to borrow anything from old solutions

The real world

- No time or budget to do green field development

Legacy software: cost and value

Legacy software is often very valuable

- the old software holds a lot of knowledge about the problem
- expensive to replace

We need some good strategies and tactics for dealing with legacy

- plan ahead: minimize discovery costs for the future
- wrappers and refactoring may help with existing software

Remember that “easy to say” doesn’t mean “easy to do”

Documentation (or lack of documentation)

How do you know anything about the functionality of your software today?

- folklore, superstition, oral tradition
- old user manuals and reference manuals
- on-line help
- web-based tutorial

The *software* is more than just code – it is a combination of the “application code” and the “know-how” of the people who use it

What if the documentation is missing?

One place to start:

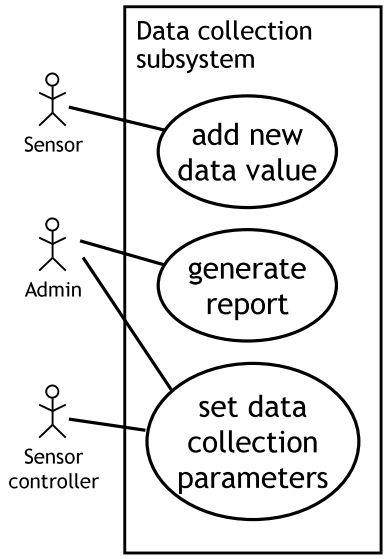
- create a “use case model” of the system – (the model is a list of participants, goals, and scenarios)
- the model doesn’t need to be complete – start to capture the most important stories

Not “algorithms + data structures = programs”

- maybe that worked OK the punch card area
- today, you need *usage scenarios* plus *data model* plus *components*

Talk to people:

- software is a “people business”



Why is legacy code difficult?

Two problems

- Discovery costs – a concept that is often overlooked
- Some requirements are “easy to say” but not “easy to do”

Discovery costs

The biggest issue for novice users of a system – discovery costs

- Discovery costs – the cost of learning what you need to know in order to do the job you have to do

What are the discovery costs of Microsoft Word?

- file system functionality in MS-Windows
- what is a document?
- fonts and colors
- lines and paragraphs
- special characters – how do I get my accent marks?
- automatic processing by MS-Word: page breaks, spelling
- navigation within a document
- section headings
- tables

How many things do you need to learn to get started?

Discovery costs

When you are making changes – think about discovery costs

Suppose you have an existing application, but it needs some changes or additions:

- Might be a new development team (unfamiliar with the design)
- Working from incomplete documentation
- Previous modifications might not be documented
- The code is the only accurate representation of the real functionality

Dealing with discovery costs

Two sets of ideas:

- How to do current software product development to reduce discovery cost issues in the future
- How to deal with old software that was designed without consideration of future discovery costs

Current software development – thinking of the future

- Documentation – what needs to be a "written tradition"?

User stories	Description of the main ways that a user will use the system – in text or pictures
Scenarios for **system configuration and management**	An outline of important administrative operations – startup, shutdown, backup, recovery
Business rules	The laws of the problem domain that the software must follow or enforce
(Optional) **Performance, quality, and security assumptions**	Important side conditions to consider any time you make a change to a system

Other ideas for reducing future discovery costs

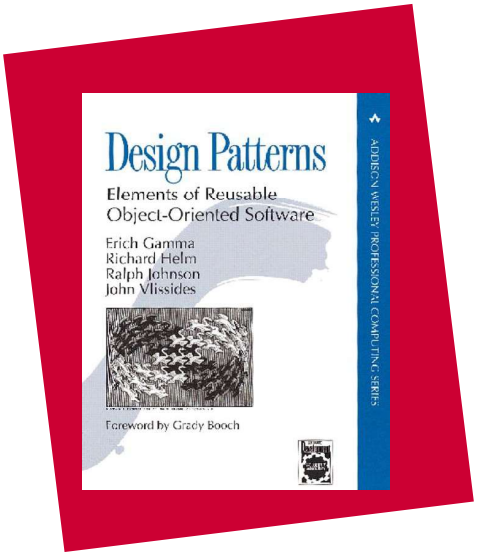
Use a programming language, databases, and design ideas that other people are likely to know

- it isn't by accident that most open-source software is written in C or Java
- using the latest tools can lead to unexpected learning curve costs for new staff members
  - new tools are OK for prototyping, applications with a short lifetime
- instead of a complex and clever design, try to use well-known design ideas – Design Patterns

Other ideas for reducing future discovery costs

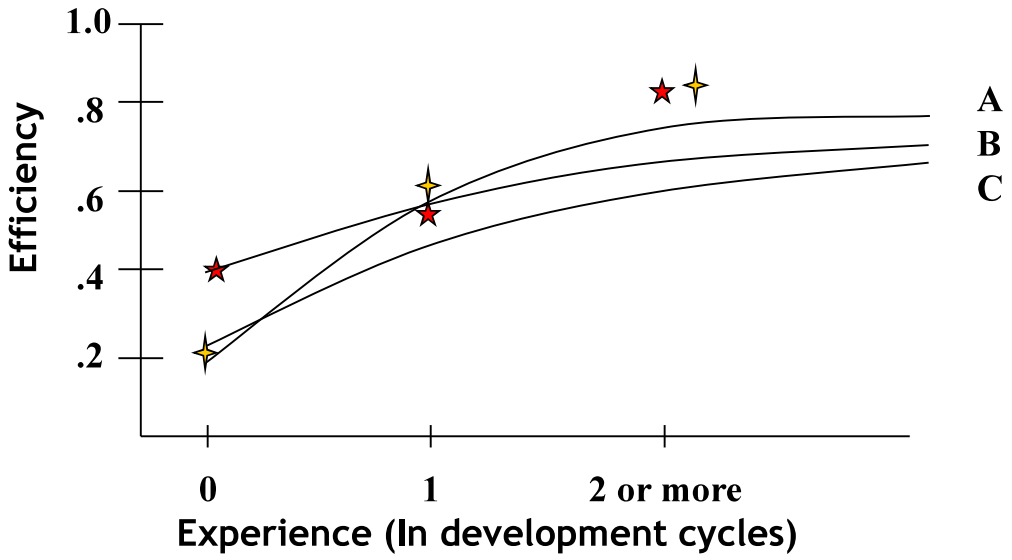
Use some standard design techniques – design patterns – to make the design more flexible

- Strategy pattern: easy to add alternative algorithms
- Composite pattern: simplify the structure of complex hierarchies of objects
- creational patterns (Factory Method, Abstract Factory, Builder, and Prototype patterns): easy to add new ways to create and initialize objects



Discovery Costs are significant

Efficiency vs. Experience (data from 3 projects)



Joseph Davison, Dennis Mancl, and William Opdyke, "Understanding and Addressing the Essential Costs of Evolving Systems," *Bell Labs Technical Journal*, April-June 2000.

What do the cost curves tell us?

Discovery costs are the dominant costs for the first two development cycles.

- In these cases, the cycles each lasted 12-18 months, associated with generic releases.

The efficiency of experienced developers (i.e. who have worked in the same area/sub-system of the same product for at least two development cycles) is three to four times that of staff who are new to that sub-system.

There is a "core" discovery component (approximately 20%) that continues, even for experienced staff.

Note: Developers who move into a new sub-system of the same product still (re-)encounter these discovery costs!

How can we live with high discovery costs?

Reengineering tactics – some ideas for working with code that wasn't designed with reuse in mind

- Try to build some of the missing documentation
- Use "OO reengineering patterns"
  - see *Object Oriented Reengineering Patterns* by Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz, http://scg.unibe.ch/download/oorp/
- Introduce unit tests into legacy code – test code is a big help for discovery
- Think about "refactoring"

Easy to say, but not easy to do

Making *bug fixes* to a finished software product is called "maintenance"

Making *new additions* to a finished software product is also called "maintenance"

- Three kinds of changes (Lientz and Swanson): adaptive, corrective, and perfective
- Which kind of change is hardest?

The surname field in the GUI needs to be at least 50 characters	The GUI must work on both Windows and Linux

How much knowledge and discovery is needed for each of these modifications?

More advice

The following items are from an OOPSLA panel “Beyond Green Field Development”:

- Reengineer using a small team, where everyone knows the roles of others
- The team should have some expertise in the legacy system
- Measure business value of the legacy system
- Apply new technologies incrementally, not all at once
- Be sure to include legacy data migration planning (not just legacy code changes)
- Be aware of economic versus emotional choices

(http://manclswx.com/workshops/oopsla03/beyond_green_report.html)

Conclusion

Legacy software is often very valuable

- the old software holds a lot of knowledge about the problem
- expensive to replace

We need some good strategies and tactics for dealing with legacy

- plan ahead: minimize discovery costs for the future
- wrappers and refactoring may help with existing software

Remember that “easy to say” doesn’t mean “easy to do”