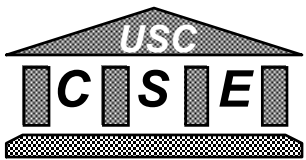


# **Agile Methods and CMM's: Oil and Water?**

**CS577B**  
**Spring 2002**



# Outline

- **The Agile Manifests**
- **Agile Example: eXtreme Programming (XP)**
  - **Counterpoint: A skeptical view**
- **Relations to CMM and CMMI**
  - **How much planning is enough?**
- **Conclusions**

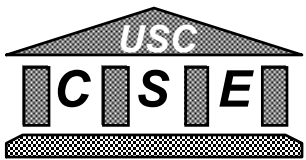
# The Agile Manifesto - I

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

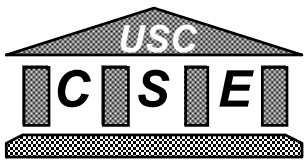
- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.



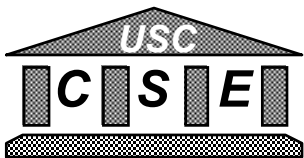
# The Agile Manifesto – II

- **Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.**
- **Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.**
- **Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.**
- **Business people and developers must work together daily throughout the project.**



# The Agile Manifesto – III

- **Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.**
- **The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.**
- **Working software is the primary measure of progress.**
- **Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.**

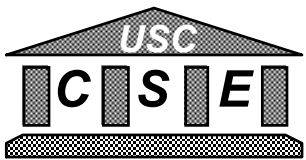


# The Agile Manifesto – IV

- **Continuous attention to technical excellence and good design enhances agility.**
- **Simplicity – the art of maximizing the amount of work not done – is essential.**
- **The best architectures, requirements, and designs emerge from self-organizing teams.**
- **At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.**

# Various Agile Methods Available

- **Adaptive Software Development (ASD)**
- **Agile Modeling**
- **Crystal methods**
- **Dynamic System Development Methodology (DSDM)**
- \* **eXtreme Programming (XP)**
- **Feature Driven Development**
- **Lean Development**
- **Scrum**



# eXtreme Programming (XP)

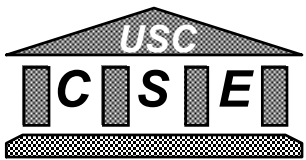
- **Principles**
- **The 12 Practices**
- **Early Adopters**

# XP Principles – I

- **Philosophy: *Take known good practices and push them to extremes***
- **“If code reviews are good, we’ll review code all the time”**
- **“If testing is good, we’ll test all the time”**
- **“If design is good, we’ll make it part of everybody’s daily business”**

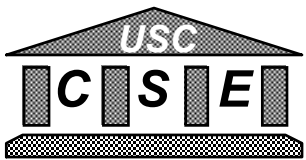
# XP Principles – II

- **“If simplicity is good, we’ll always leave the system with the simplest design that supports its current functionality”**
- **“If architecture is important, everybody will work defining and refining the architecture all the time”**
- **“If integration testing is important, then we’ll integrate and test several times a day”**



# XP Principles – III

- **“If short iterations are good, we’ll make the iterations really, really short – seconds and minutes and hours, not weeks and months and years”**
- **“If customer involvement is good, we’ll make them full-time participants”**



# XP: The 12 Practices

- **The Planning Game**
- **Small Releases**
- **Metaphor**
- **Simple Design**
- **Testing**
- **Refactoring**
- **Pair Programming**
- **Collective Ownership**
- **Continuous Integration**
- **40-hour Week**
- **On-site Customer**
- **Coding Standards**

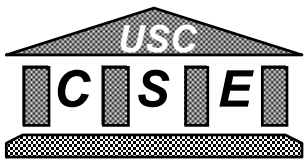
**-Used generatively, not imperatively**

# The Planning Game

- **Use stories to facilitate knowledge transfer**
- **Put decisions in the hands of the person with the best knowledge:**
  - **business decisions → Customer**
  - **software decisions → Developer**
- **Plan only as far as your knowledge allows**
  - **next iteration or next release**

# Small Releases

- **Supports quick feedback from users**
- **Simplify the tracking of metrics**
  - **stories per iteration → project velocity**
- **Increase the manageability of the project for the customer**
  - **But complicate user conservation of familiarity**



# Metaphor

- **Ground all discussions in a single shared story of how the whole system works**
- **Provide an overarching view of the project**
- **Connect program to work process**

# Simple Design

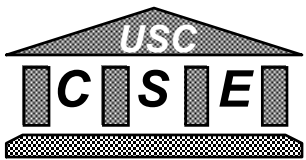
- **Design Embodies only the needed complexity and no more**
  - emphasis on top-down or bottom-up design as needed to meet this iteration's stories
  - extra complexity removed when discovered
- **Simpler designs are easier to modify, maintain, and describe**
  - decreases the cost of design changes
  - **But no notion of product line architecture**

# Testing

- **Unit tests verify the programmer's work**
  - must be done by programmer
  - constant testing makes finding new bugs faster and easier
- **Functional tests verify that a story is complete**
  - developed by customer
  - tests define functional requirements

# Refactoring

- **Procedure for implementing iterative design**
  - behavior-preserving
  - improves communication among developers
  - adds flexibility to the programming process
- **Design is important – do it all the time**
  - software development process is a design process
  - But redesign much more expensive for large systems

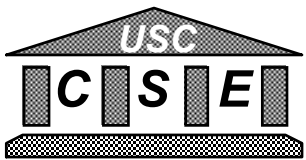


# Pair Programming

- **All code is written by two programmers at a single machine**
- **Inspections are important, so do them all the time**
- **Increase implicit knowledge transfer**
- **Decrease cycle time, but increase effort**

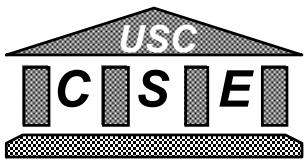
# Collective Ownership

- **Everyone owns all of the code**
  - anyone can change any code anywhere
  - no personal ownership of modules
  - no egoless programming either
- **Everyone is permitted access to all the code so everyone has a stake in knowing all of the code (that they will work with)**
- **Requires deserved trust**
  - But still has scalability problems



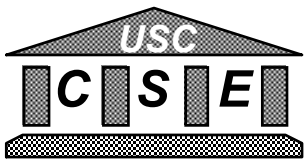
# Continuous Integration

- **The system always works**
  - there is always something to be released
- **Similar to rapid releases**
  - fast feedback to developers on problems
  - no ‘big bang’ integration disasters



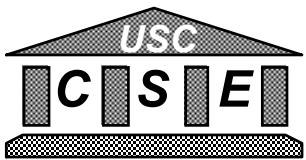
# 40-hour Week

- **No heroes**
- **Knowledge can only be transferred at a limited rate**
- **Work for sustained speed, not a single sprint**
  - **never work overtime a second week in a row**



# On-site Customer

- **A real, live user available full-time to answer questions as they occur**
- **Programmers don't know everything**
- **Business knowledge is the key to a successful business project**



# Coding Standards

- **Communication occurs through the code**
- **Common standard promotes understanding of other developers' code**
- **Helps promote team focus**



# Counterpoint: A Skeptical View – I

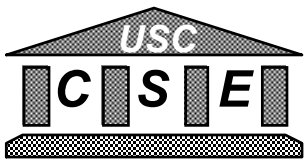
## Letter to Computer, Steven Rakin, Dec. 2000

**“individuals and interactions over processes and tools”**

Translation: Talking to people gives us the flexibility to do whatever we want in whatever way we want to do it. Of course, it's understood that we know what you want - even if you don't.

**“working software over comprehensive documentation”**

Translation: We want to spend all our time coding. Real programmers don't write documentation.



## **Counterpoint: A Skeptical View – II**

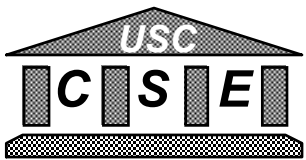
### **Letter to Computer, Steven Rakin, Dec. 2000**

#### **“customer collaboration over contract negotiation”**

Translation: Let's not spend time haggling over the details, it only interferes with our ability to spend all our time coding. We'll work out the kinks once we deliver something...

#### **“responding to change over following a plan”**

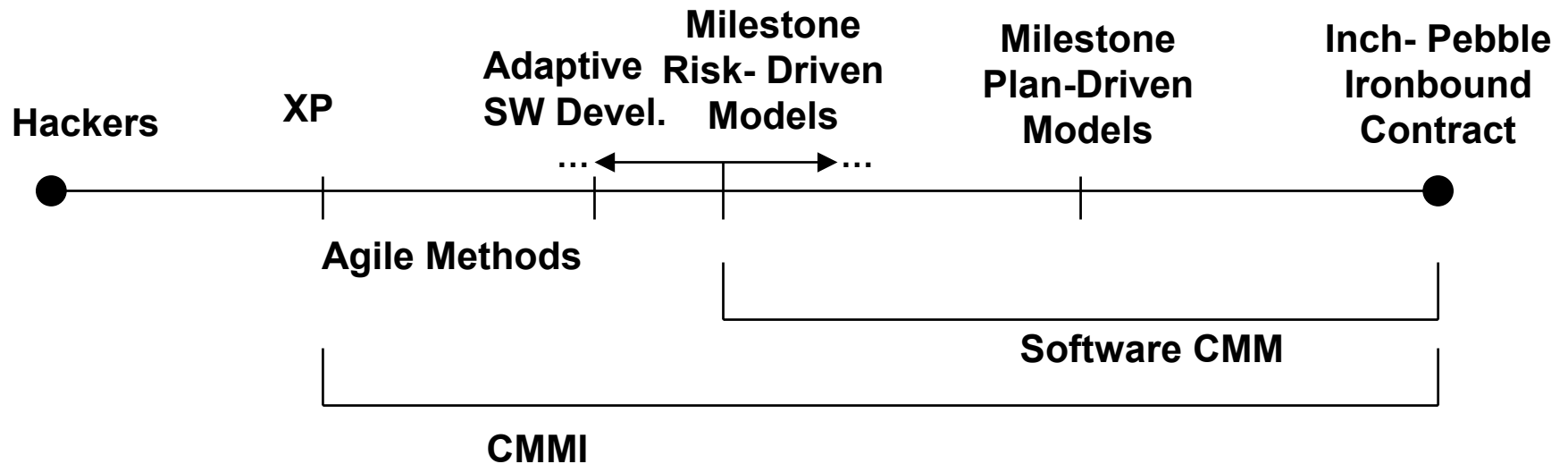
Translation: Following a plan implies we would have to spend time thinking about the problem and how we might actually solve it. Why would we want to do that when we could be coding?



# Outline

- **The Agile Manifests**
- **Agile Example: eXtreme Programming (XP)**
  - **Counterpoint: A skeptical view**
- • **Relations to Software CMM and CMMI**
  - **The planning spectrum**
  - **Agile and Plan-Driven home grounds**
  - **How much planning is enough?**
- **Conclusions**

# The Planning Spectrum



# Agile and Plan-Driven Home Grounds

## Agile Home Ground

- Agile, knowledgeable, collocated, collaborative developers
- Dedicated, knowledgeable, collocated, collaborative, representative, empowered customers
- Largely emergent requirements, rapid change
- Architected for current requirements
- Refactoring inexpensive
- Smaller teams, products
- Premium on rapid value

## Plan-Driven Home Ground

- Plan-oriented developers; mix of skills
- Mix of customer capability levels
- requirements knowable early; largely stable
- Architected for current and foreseeable requirements
- Refactoring expensive
- Larger teams, products
- Premium on high-assurance

# How Much Planning Is Enough?

## - A risk analysis approach

- **Risk Exposure RE = Prob (Loss) \* Size (Loss)**

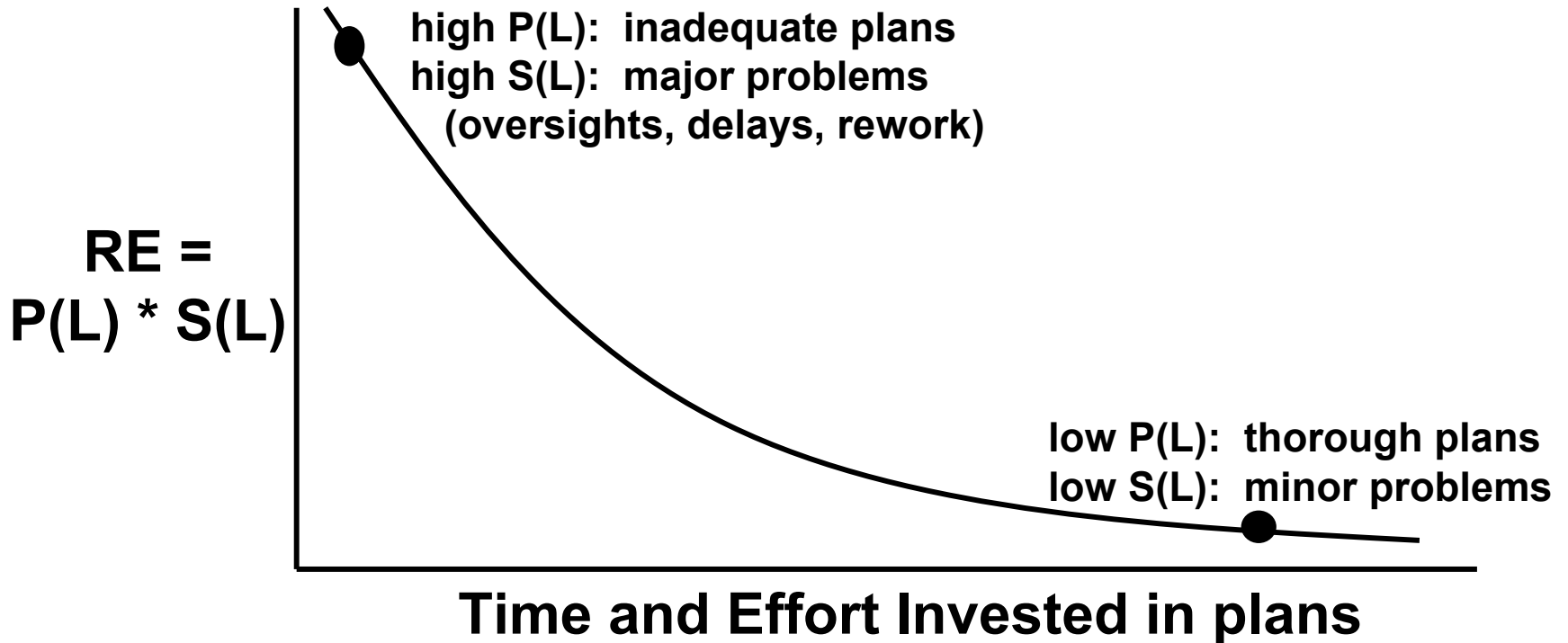
- “Loss” – financial; reputation; future prospects, ...

- **For multiple sources of loss:**

$$RE = \sum_{\text{sources}} [\text{Prob (Loss)} * \text{Size (Loss)}]_{\text{source}}$$

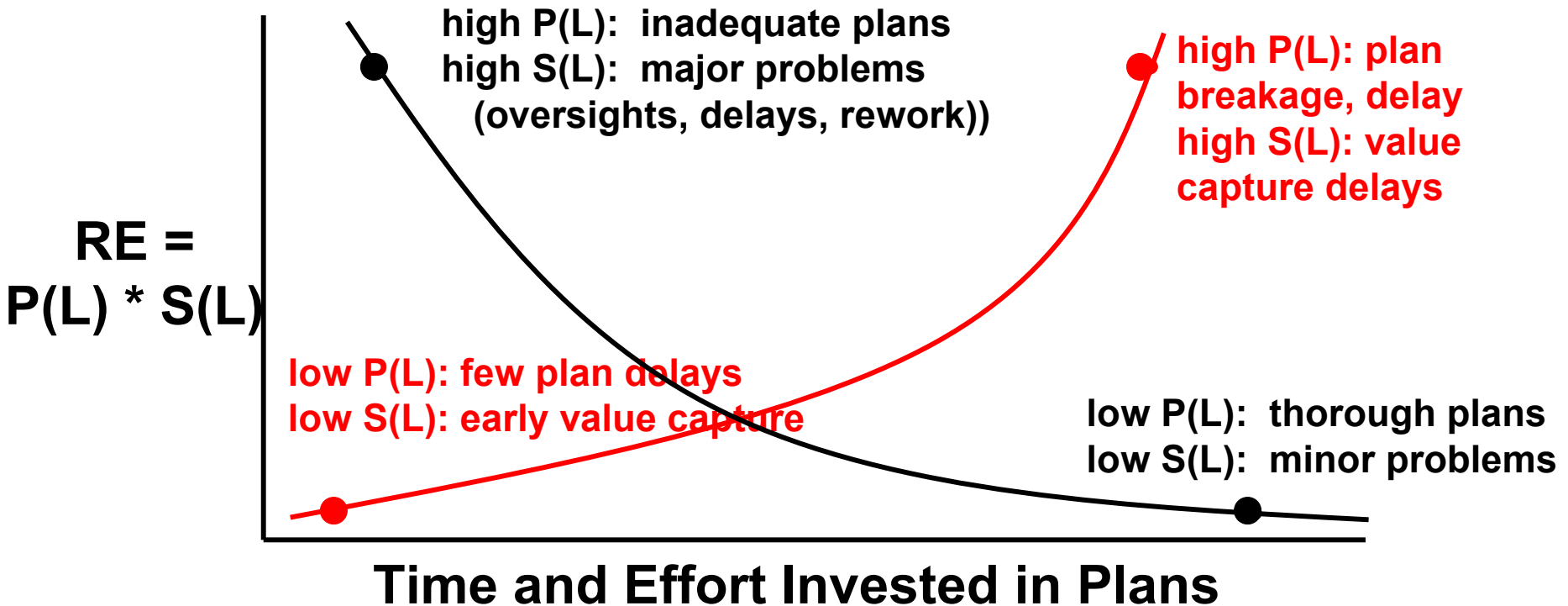
# Example RE Profile: Planning Detail

- Loss due to inadequate plans



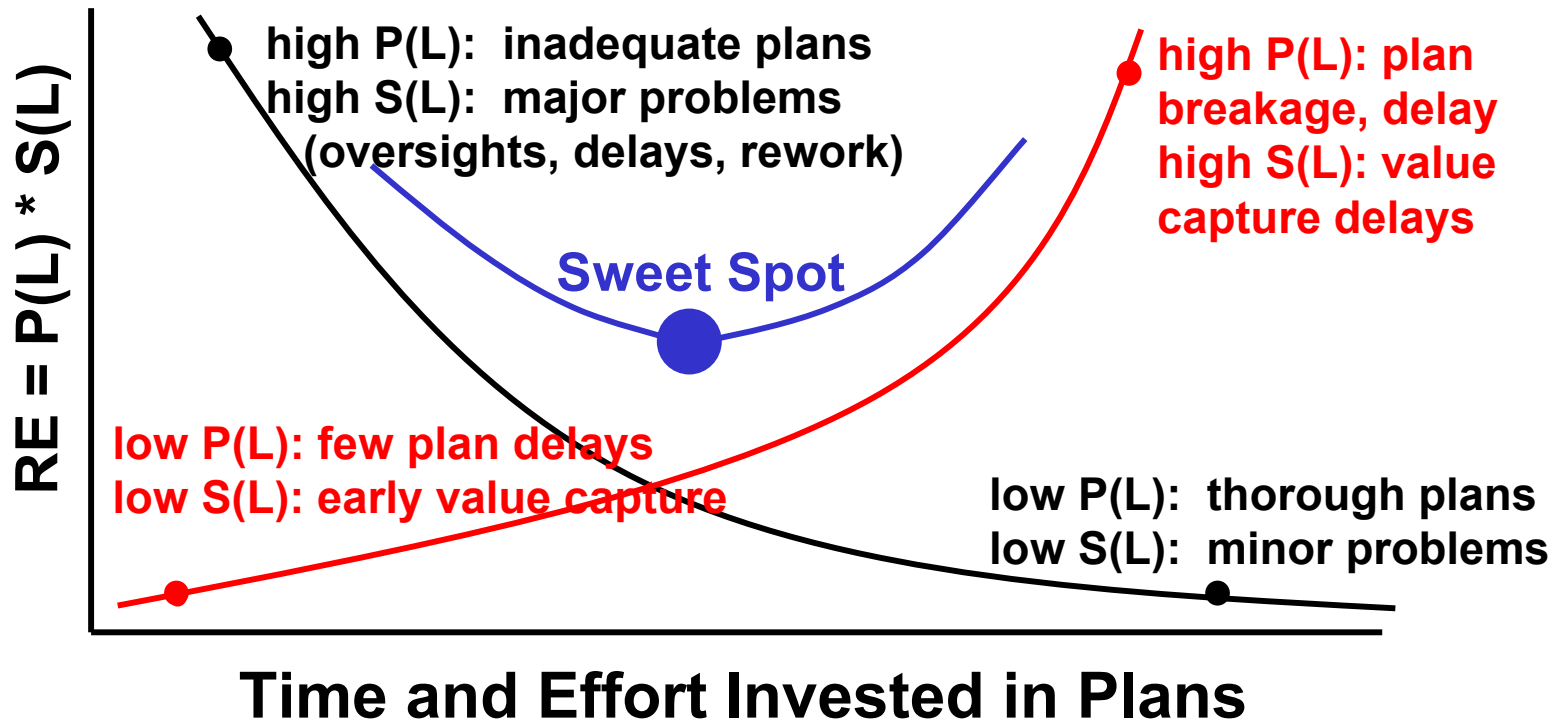
# Example RE Profile: Planning Detail

- Loss due to inadequate plans
- **Loss due to market share erosion**

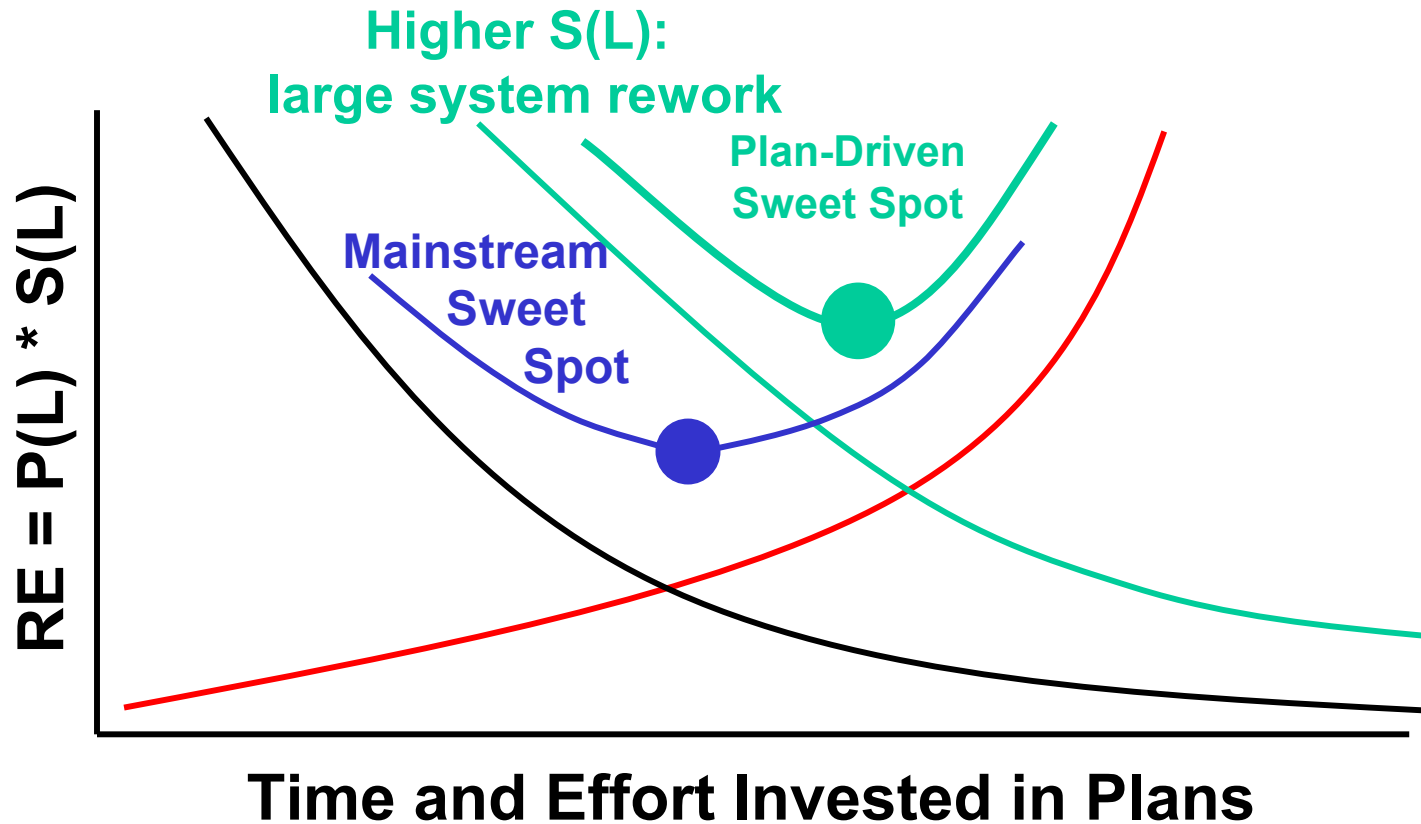


# Example RE Profile: Planning Detail

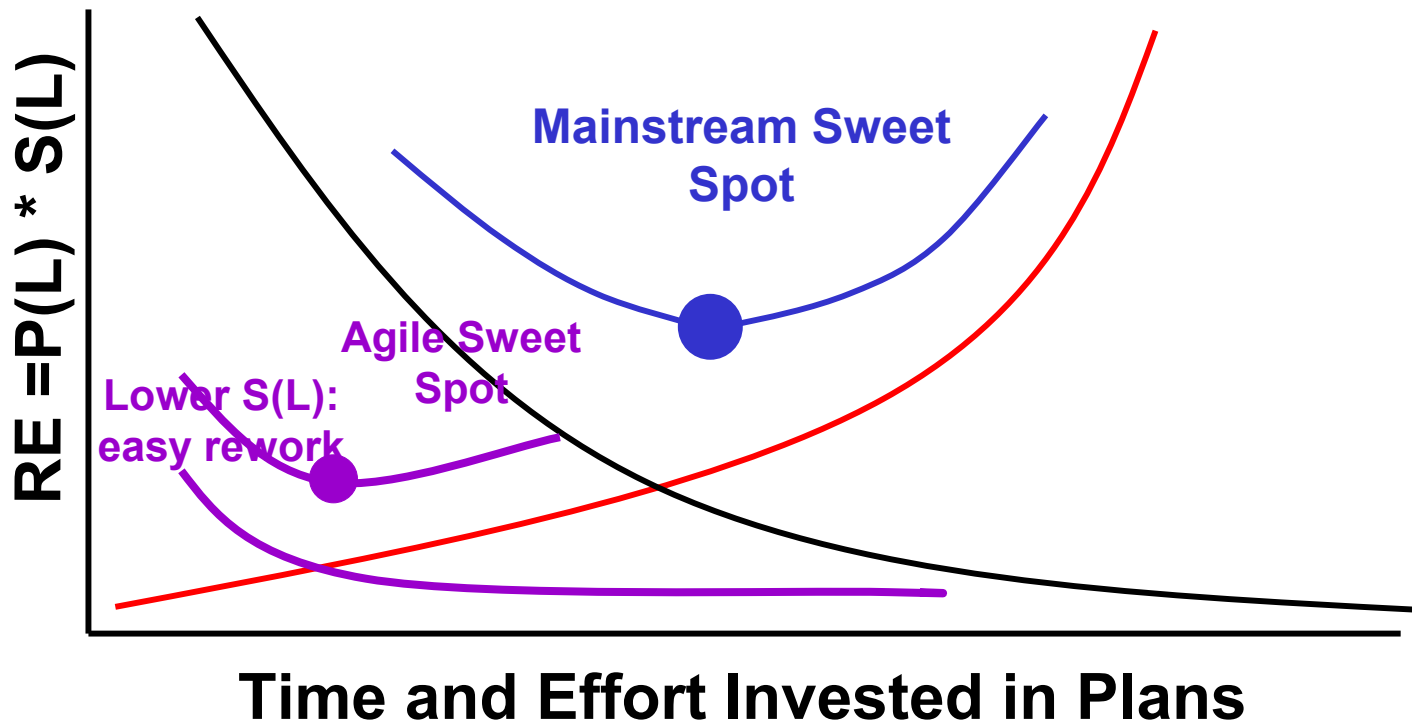
## - Sum of Risk Exposures



# Comparative RE Profile: Plan-Driven Home Ground



# Comparative RE Profile: Agile Home Ground



# Conclusions: CMMI and Agile Methods

- **Agile and plan-driven methods have best-fit home grounds**
  - Increasing pace of change requires more agility
- **Risk considerations help balance agility and planning**
  - Risk-driven “How much planning is enough?”
- **Risk-driven agile/plan-driven hybrid methods available**
  - Adaptive Software Development, RUP, MBASE, CeBASE Method
- **CMMI provides enabling criteria for hybrid methods**
  - Risk Management, Integrated Teaming