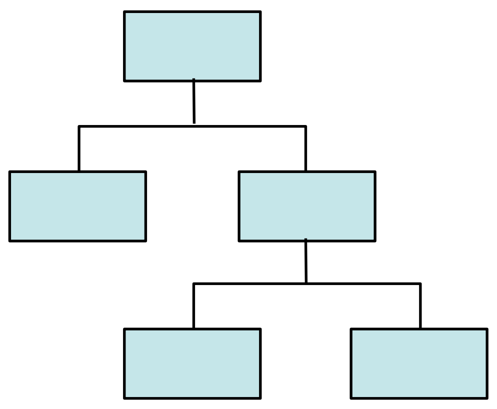
Tecnologia e Applicazioni Internet 2009/10

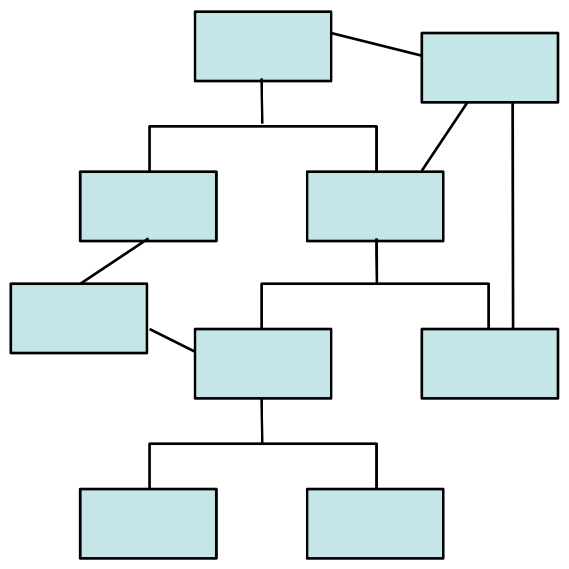
Lezione 1 - Il principio aperto/chiuso

Matteo Vaccari
http://matteo.vaccari.name/
vaccari@pobox.com

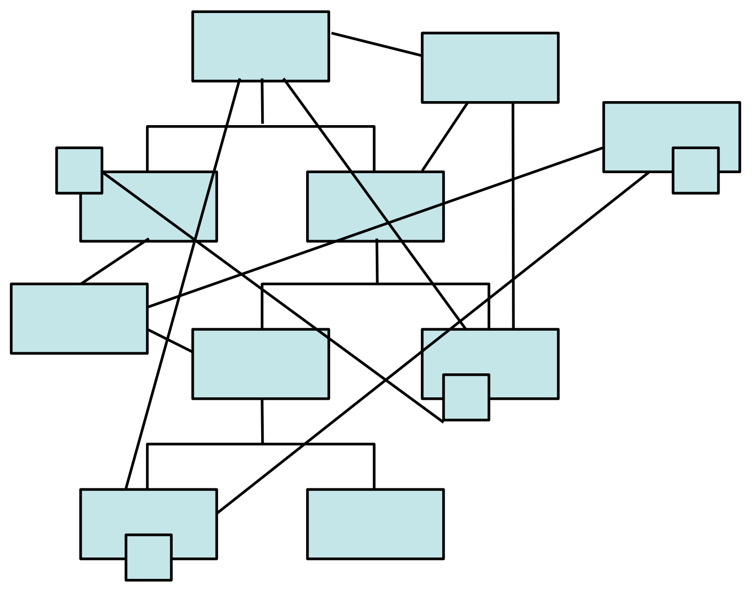
Che cos’è il *debito tecnico?*



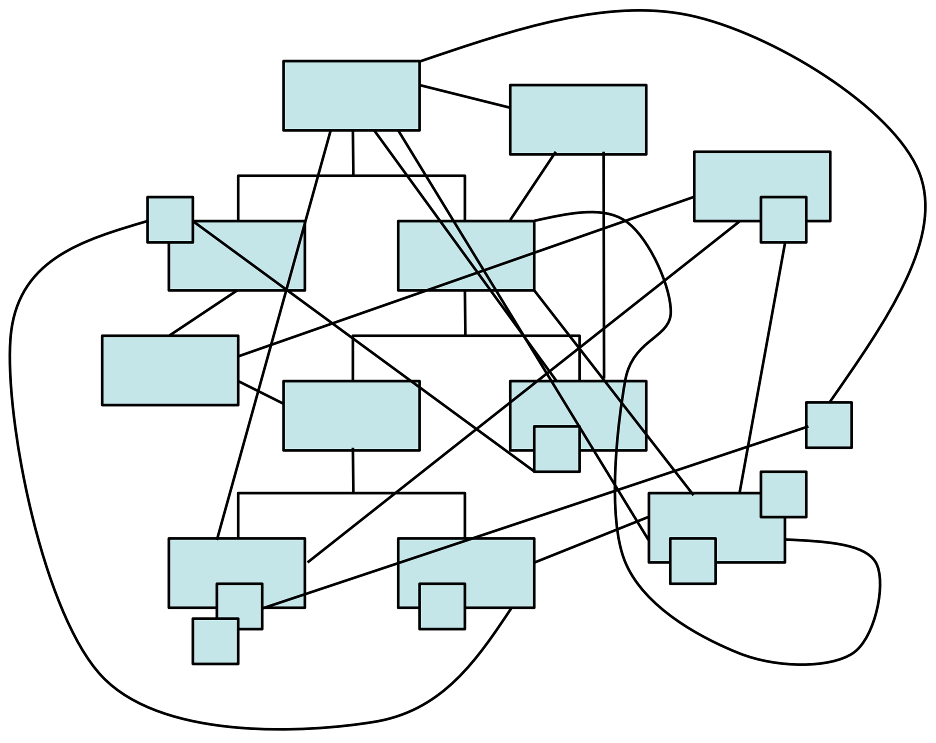
Cost of change: C



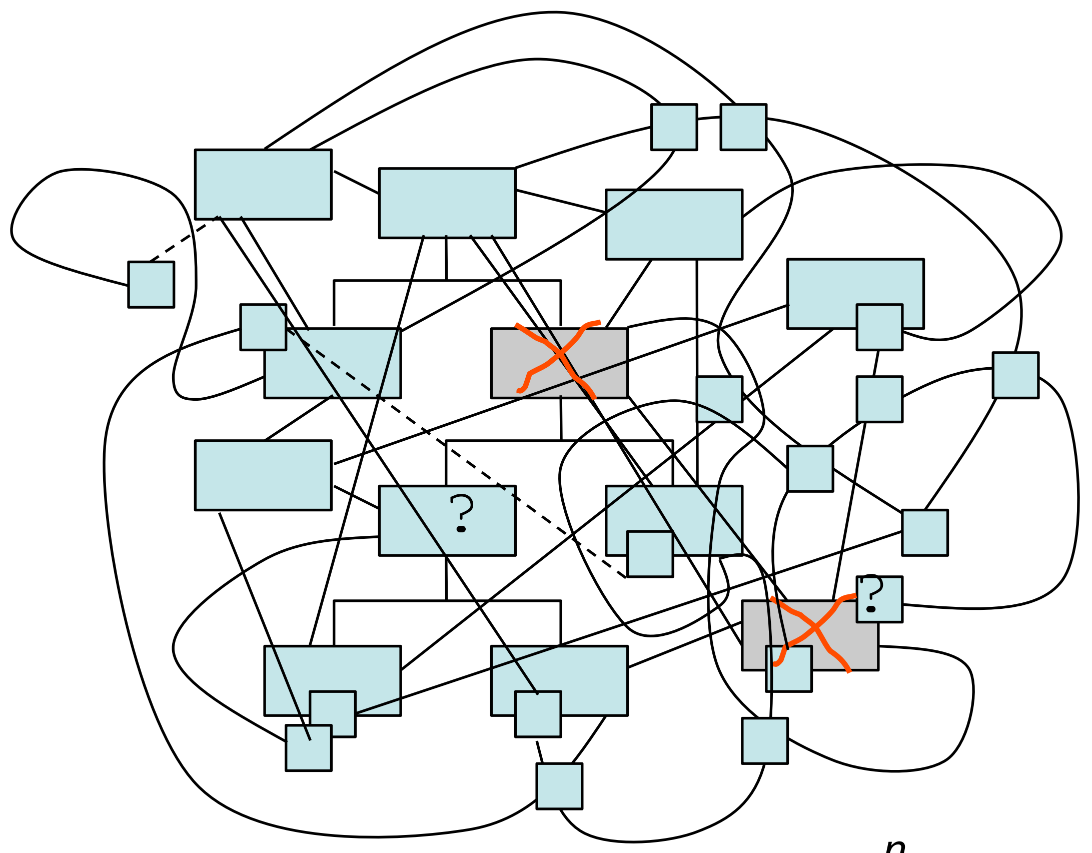
Cost of change: C + *n*



Cost of change: C x *n*



Cost of change: C^{n}



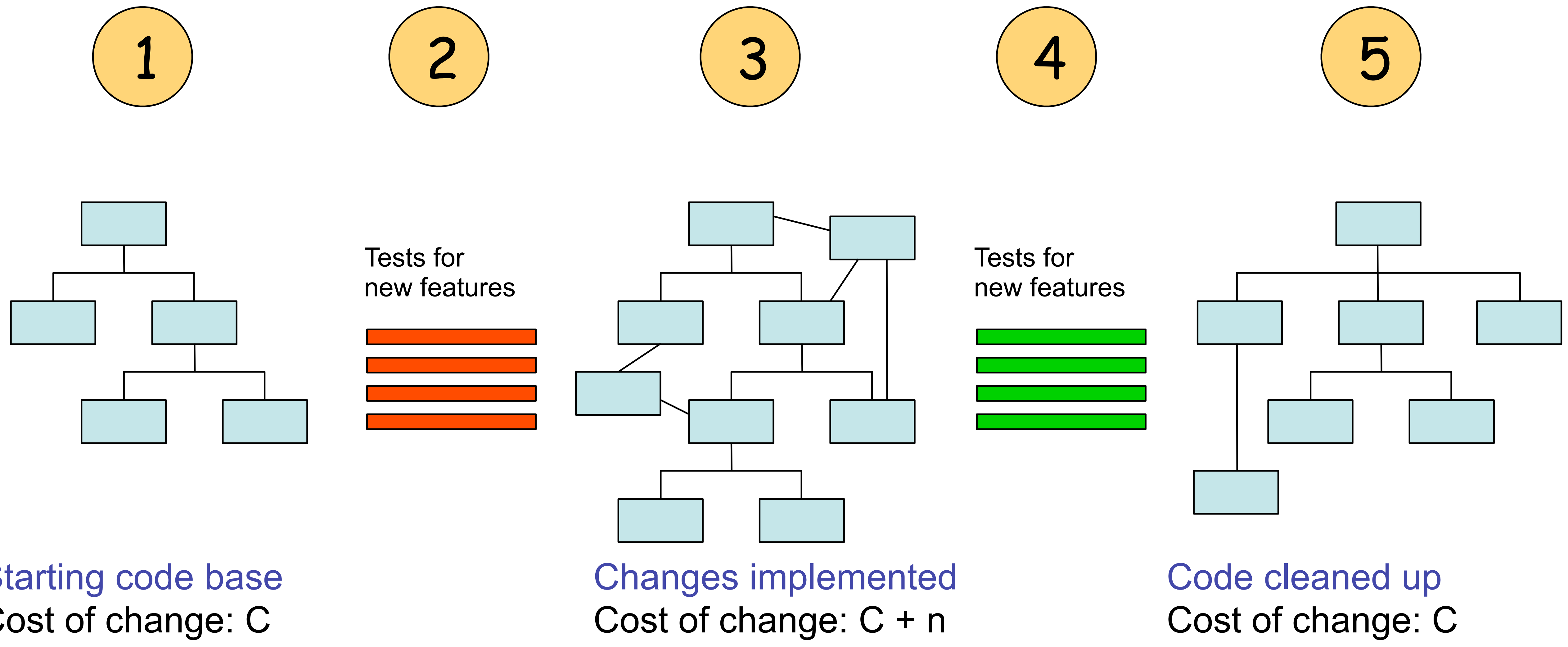
Cost of change: $C^{n^{n}}$

The little idealized class models appear one by one with mouse clicks.

The concept illustrated here is that the cost of changing the software increases as the “messiness” of the system design (as expressed in the living code, not necessarily in a design document) increases. Eventually software becomes so hard to understand and so risky to modify that it is no longer worth the effort to enhance it. This is what happens when you allow design debt to pile up over time.

Slides by *Dave Nicolette*

Pagare le *rate* del debito tecnico

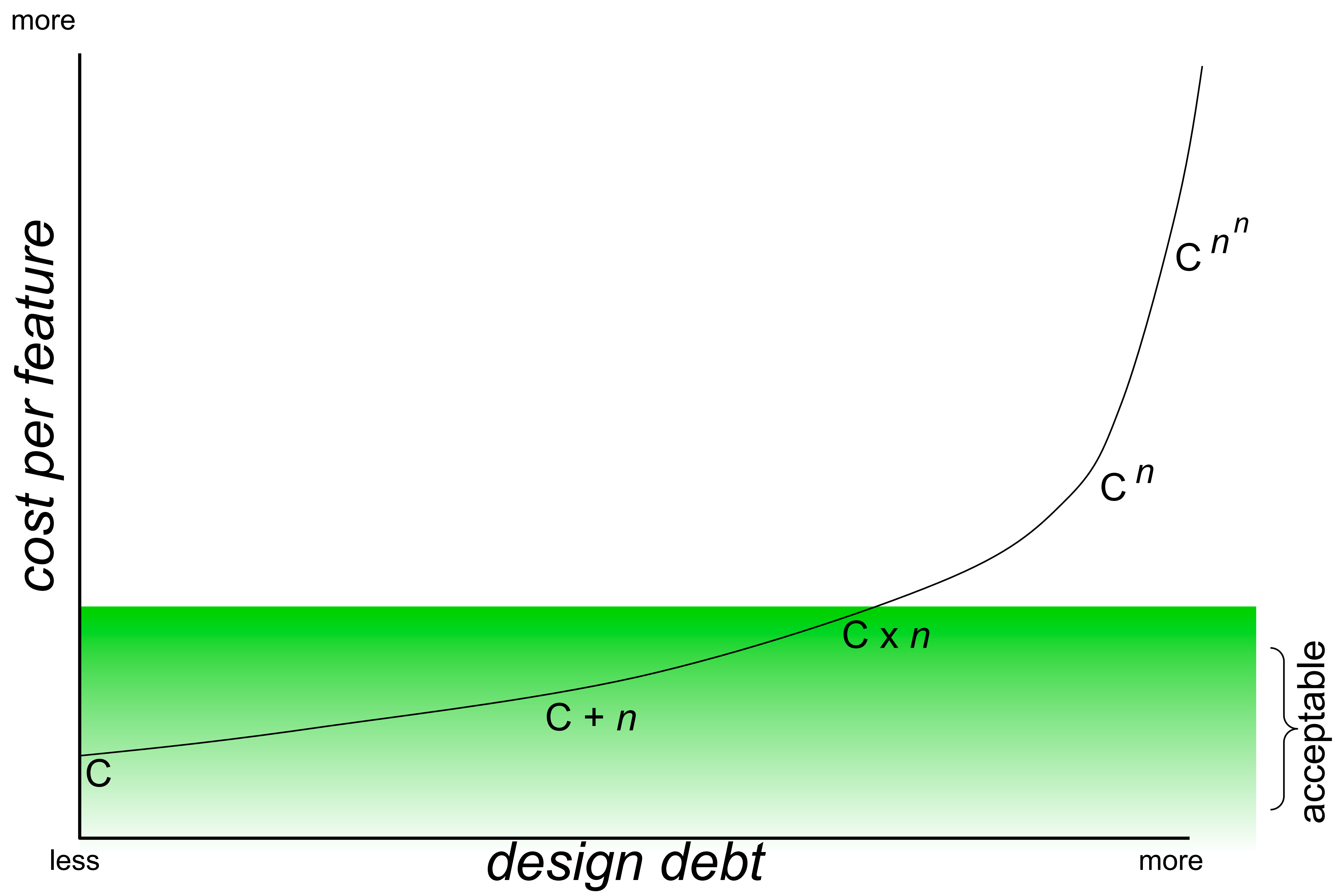


The slide comes up with step 1 visible. The remaining steps are displayed one by one by mouse clicks.

This is how TDD alleviates the problem of “compound interest” in design debt. By working in tight red-green-refactor loops, design debt is paid off in small installments throughout the development process. We never allow “interest” to accumulate to a high level. You might want to avoid the word “refactor” depending on the audience. “Cleaned up code” might communicate the concept more intuitively.

Slides by *Dave Nicolette*

Effect of design debt on development cost during a project

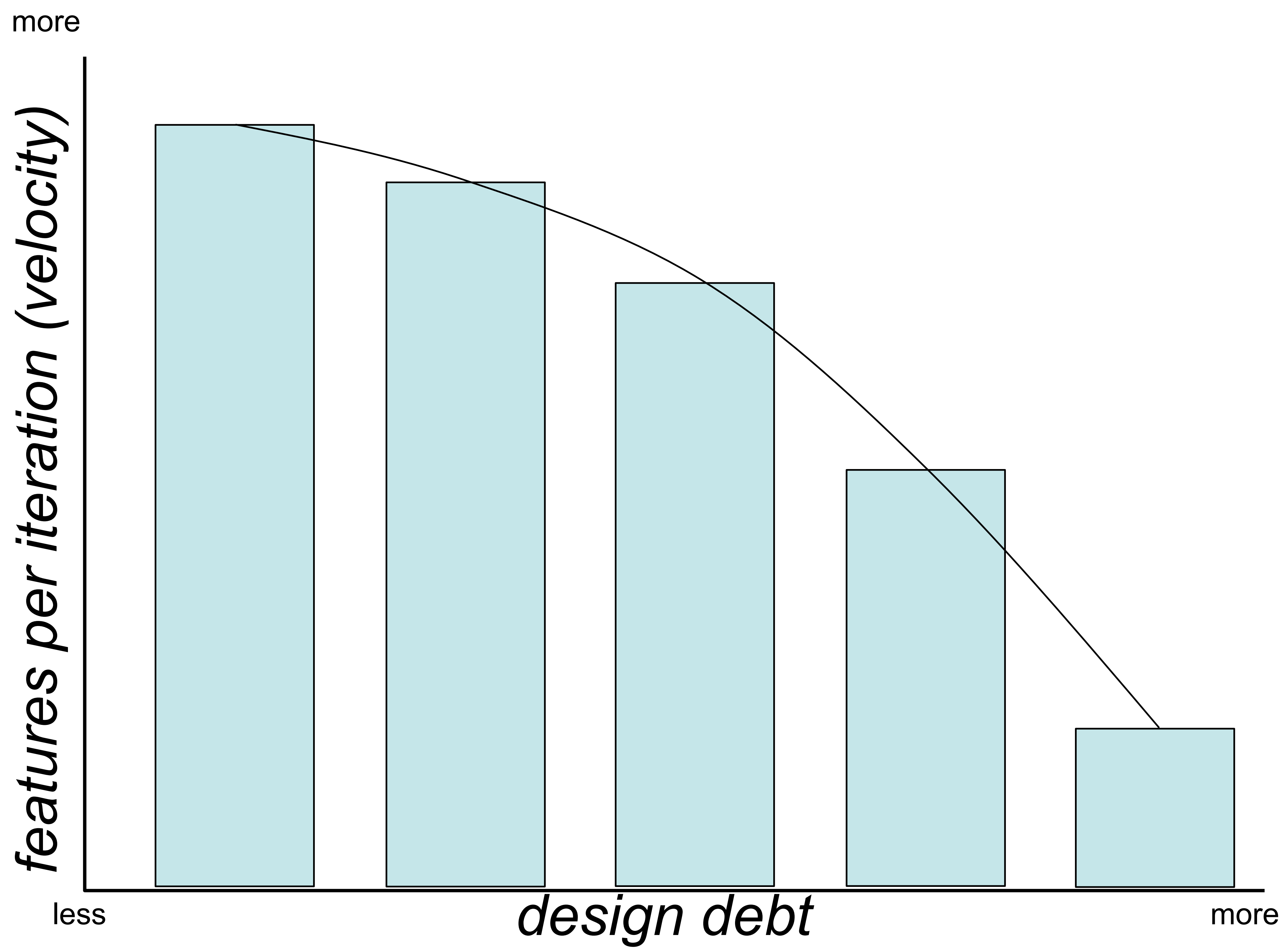


Now we want to explain why the managers/customers/product owners care about the concept of design debt. They may still see it as a technical issue. The next several slides illustrate the effect of mounting design debt on the cost of changing a system, the quality of a system, the cost of supporting a system in production, and the useful lifetime of a system.

This slide shows the cost of change that was described in an earlier slide, but as a trend line. As design debt mounts, the cost of adding features to a system rises to an untenable level.

Slides by *Dave Nicolette*

Effect of design debt on team productivity during a project



Design debt reduces the productivity of the development team because as the size and complexity of the solution increase, a messy (living) design makes every modification more difficult. They must spend additional time analyzing the potential effect of any change they propose to make. They can easily overlook a piece of code that should be changed, thus creating a defect. They can easily break functionality that was working, even if that functionality is unrelated to the enhancement they are making.

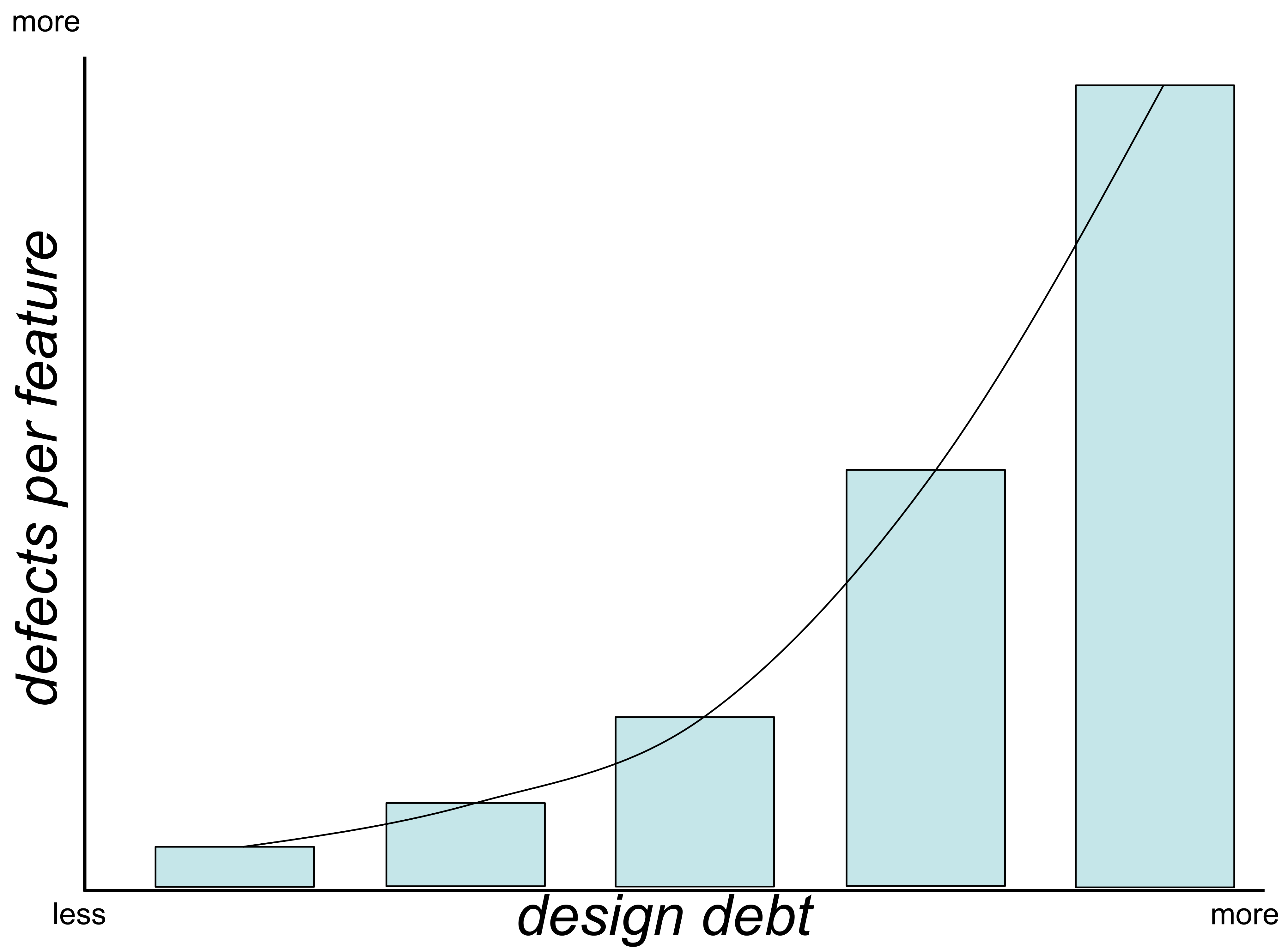
Testing also takes more time because it is harder to construct reliable and meaningful test cases for messy code, and easier for different test cases to interfere with each other because of implementation details that “should not matter” based on the purpose of the new test case.

Therefore it is not only productivity that suffers, but solution quality as well.

Another cost to the customer is that the additional time developers spend working with the messy code base is time they cannot spend designing, building, and testing features that add value to the solution. That means fewer features in the end product.

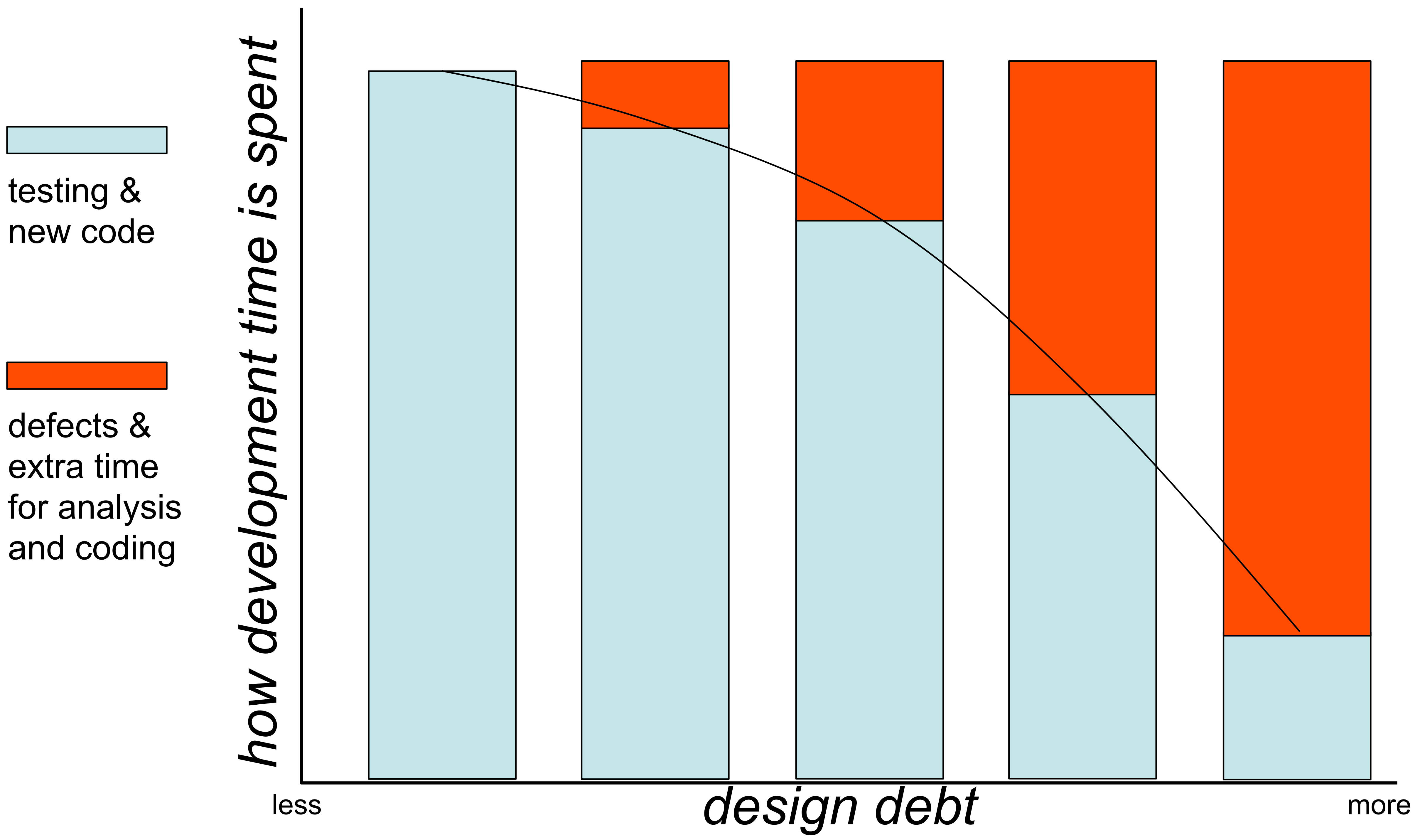
Slides by *Dave Nicolette*

Effect of design debt on the defect rate during a project



Due to some of the issues mentioned on the preceding slide, design debt tends to increase the rate of production of defects. As a solution grows in size and complexity, it is normal to see an increase in the absolute number of defects over the course of a project. However, the rate of creation of defect should remain constant. If the rate trends upward, then the development work will end up consisting almost entirely of defect correction and there will be little time left to develop value-add features.

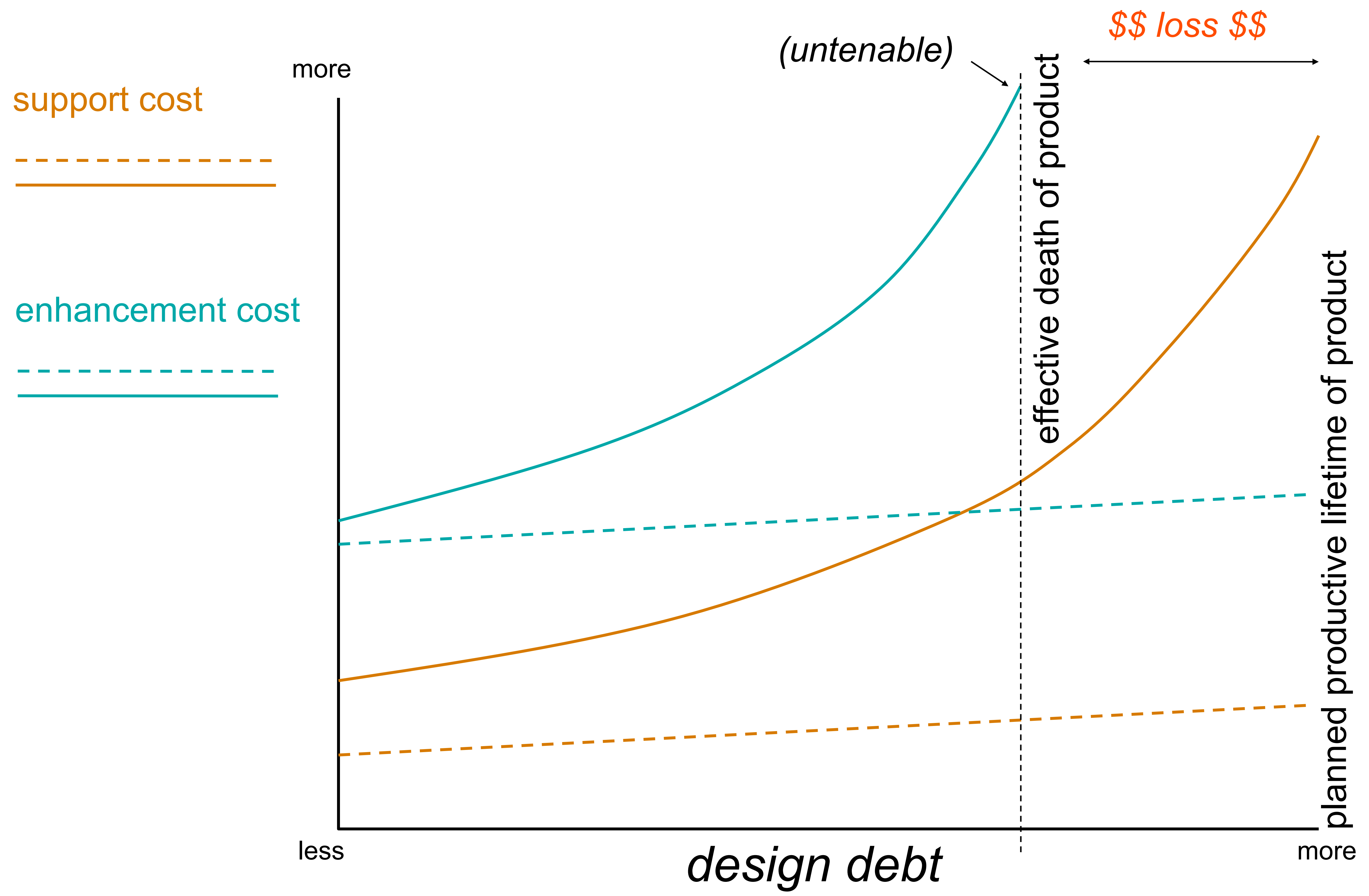
Effect of design debt on time available for testing during a project



This is another way to depict the effect of design debt on the productive time available to the development team. The red areas are meant to illustrate time that has to be spent “working around” the messy code base. This time would be available for productive work if the design debt had been paid down incrementally throughout the project.

Slides by *Dave Nicolette*

Effect of design debt over the life of a product



This slide displays various elements one by one with mouse clicks. At first you will see the chart area and labels.

Support cost is displayed next. The dotted line depicts planned support cost and the solid line depicts actual cost. Messy code tends to have more defects and the defects are harder to correct for reasons mentioned previously.

Enhancement cost is displayed next. This represents the cost of projects undertaken to enhance the solution after it has been in production. Another click will display the information about the “effective death of the product.” The point to make here is that long-term strategic planning and budgeting assume the product will have a certain useful production lifetime. The effect of design debt on the maintainability of the solution means that lifetime will be shorter than anticipated. This represents needless additional costs that can be extremely high, depending on the size and criticality of the solution, and on ancillary effects on other aspects of the business plan.

The small matter of whether a development team on a particular project uses TDD rigorously or not can have a “ripple” or “domino” effect that has implications on a far larger scale. Consider what would happen in an enterprise in which 200 or 300 products all “died” prematurely, as the slide depicts.

Che aspetto ha il *debito tecnico?*

```
IRequestCodingStrategy requestCodingStrategy = requestCycle.getProcessor()
        .getRequestCodingStrategy();

final String path = requestParameters.getPath();
IRequestTarget target = null;

// See whether this request points to a bookmarkable page
if (requestParameters.getBookmarkablePageClass() != null)
{
        target = resolveBookmarkablePage(requestCycle, requestParameters);
}
// See whether this request points to a rendered page
else if (requestParameters.getComponentPath() != null)
{
        // marks whether or not we will be processing this request
        int processRequest = 0; // 0 == process, 1 == page expired, 2 == not active page anymore
        synchronized (requestCycle.getSession())
        {
                // we need to check if this request has been flagged as
                // process-only-if-path-is-active and if so make sure this
                // condition is met
                if (requestParameters.isOnlyProcessIfPathActive())
                {
                        // this request has indeed been flagged as
                        // process-only-if-path-is-active

                        Session session = Session.get();
                        IPageMap pageMap = session.pageMapForName(requestParameters.getPageMapName(), false);
                        if (pageMap == null)
                        {
                                // requested pagemap no longer exists - ignore this
                                // request
                                processRequest = 1;
                        }
                        else if (pageMap instanceof AccessStackPageMap)
                        {
                                AccessStackPageMap accessStackPageMap = (AccessStackPageMap)pageMap;
                                if (accessStackPageMap.getAccessStack().size() > 0)
                                {
                                        final Access access = (Access)accessStackPageMap.getAccessStack()
                                                .peek();

                                        final int pageId = Integer.parseInt(Strings.firstPathComponent(
```

Il *FizzBuzz*

**1, 2, Fizz!, 4, Buzz!, Fizz!, 7,
8, Fizz!, Buzz!, 11, Fizz!, 13,
14, FizzBuzz!, 16, 17, Fizz!...**

Se è divisibile per 3, dì "Fizz!"
Se è divisibile per 5, dì "Buzz!"
Se è divisibile per 3 e 5, dì "FizzBuzz!"
Altrimenti dì il numero.

Non è difficile...

```java
public String say(Integer n) {
    if (isFizz(n) && isBuzz(n)) {
        return "FizzBuzz";
    }
    if (isFizz(n)) {
        return "Fizz";
    }
    if (isBuzz(n)) {
        return "Buzz";
    }
    return n.toString();
}

public boolean isFizz(Integer n) {
    return 0 == n % 3;
}

// ...
```

Nuovo requisito

Se è divisibile per 7, dì “Bang!”

Nessun problema!

```java
public String say(Integer n) {
    if (isBang(n)) {
        return "Bang";
    }
    if (isFizz(n) && isBuzz(n)) {
        return "FizzBuzz";
    }
    if (isFizz(n)) {
        return "Fizz";
    }
    if (isBuzz(n)) {
        return "Buzz";
    }
    return n.toString();
}
```

Non è così che intendevo...

Se è divisibile per 3 e per 7, dì “FizzBang!”
Se è divisibile per 5 e per 7, dì “BuzzBang!”
Se è divisibile per 3, 5 e 7, dì “FizzBuzzBang!”

Hmmm....

```java
public String say(Integer n) {
    if (isFizz(n) && isBuzz(n) && isBang(n)) {
        return "FizzBuzzBang";
    }
    if (isBang(n) && isBuzz(n)) {
        return "BuzzBang";
    }
```

Non è più così semplice. Che cosa succederà al prossimo cambio di requisiti?

```java
    if (isFizz(n) && isBuzz(n)) {
        return "FizzBuzz";
    }
    if (isFizz(n)) {
        return "Fizz";
    }
    if (isBuzz(n)) {
        return "Buzz";
    }
    return n.toString();
}
```

OK. Nessuno ve l’ha detto finora ma...

Aggiungere IF è il **male**.

http://pierg.wordpress.com/2009/08/05/anti-if-campaign/

COMODO ≠ EFFICACE

http://www.antiifcampaign.com/

Il principio *aperto/chiuso*

Software entities
(classes, modules, functions, etc.)
should be *open for extension*, but
closed for modification

Demo -- come risolvere il FizzBuzz con gli oggetti

Quando si deve estendere la funzionalità del sistema:

- Posso farlo modificando *solo la costruzione* ed eventualmente creando *nuove classi*?
- Se sì, bene! ➪ €€€€
- Se no, *rifattorizzo* fino a quando non posso

Refactoring:

Safely improve the design of existing code

Refactoring:

Safely improve the design of existing code

does not add functionality

Refactoring:

Safely improve the design of existing code

not rewriting from scratch

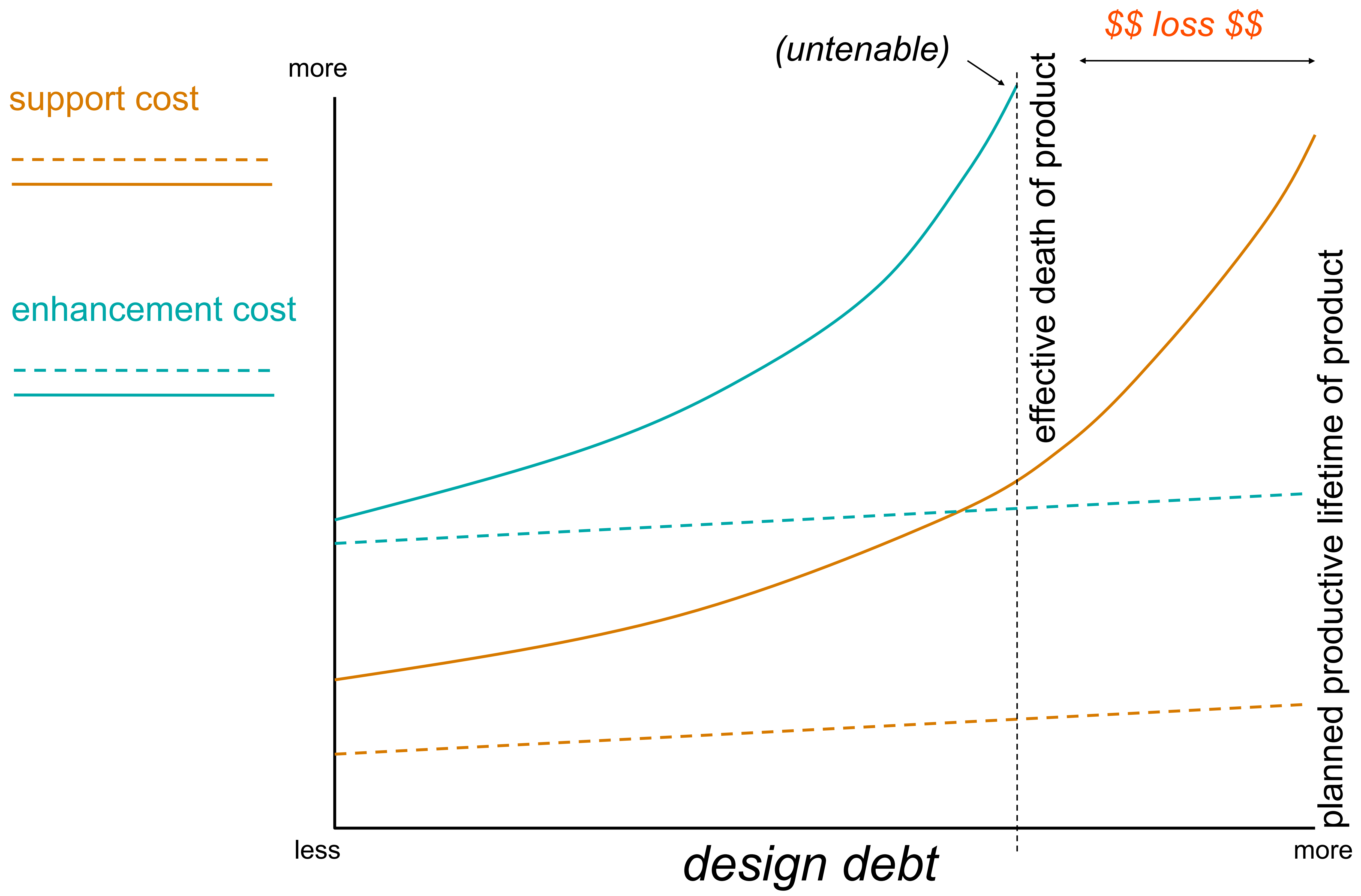
Refactoring:

Safely improve the design of existing code

baby steps

tests

Perché fare refactoring?



Che cosa rende il codice *difficile da testare?*

Mescolare *new* e *logica*

```
@Test
public void shouldRecognizeExpiredOption() {
    Date expiration = dateAt(1995, MAY, 28);

    Option option = new Option(expiration);

    assertTrue(option.isExpired());
}
```

```
public class Option {

    private final Date expiration;

    public Option(Date expiration) {
        this.expiration = expiration;
    }

    public boolean isExpired() {
        Date now = new Date();
        return expiration.before(now);
    }
}
```

Roberto Albertini, Sourcesense

```java
@Test
public void shouldRecognizeExpiredOption() {
    Date expiration = dateAt(2008, MAY, 28);

    Option option = new Option(expiration);

    assertTrue(option.isExpired());
}
```

```java
public class Option {

    private final Date expiration;

    public Option(Date expiration) {
        this.expiration = expiration;
    }

    public boolean isExpired() {
        Date now = new Date();
        return expiration.before(now);
    }
}
```

```java
@Test
public void shouldRecognizeNonExpiredOption() {
    Date expiration = dateAt(2020, MAY, 28);

    Option option = new Option(expiration);

    assertFalse(option.isExpired());
}
```

Per quanto tempo funzionerà?

Come posso testare i casi limite?

Roberto Albertini, Sourcesense

new Date() non è programmabile

```
public boolean isExpired() {
  Date now = new Date();
  return expiration.before(now);
}
```

lo sostituisco con un collaboratore

```
public boolean isExpired() {
  Date now = clock.now();
  return expiration.before(now);
}
```

Roberto Albertini, Sourcesense

```java
private final Date expiration;

public Option(Date expiration) {
    this.expiration = expiration;
}
```

devo esplicitare la dipendenza rendere iniettabile il collaboratore

```java
private final Date expiration;
private final Clock clock;

public Option(Date expiration, Clock clock) {
    this.expiration = expiration;
    this.clock = clock;
}
```

Il codice di produzione

```java
public class Option {

    private final Date expiration;
    private final Clock clock;

    public Option(Date expiration, Clock clock) {
        this.expiration = expiration;
        this.clock = clock;
    }

    public Option(Date expiration) {
        this(expiration, new RealClock());
    }
}
```

```java
public interface Clock {
    Date now();
}
```

```java
public class RealClock implements Clock {

    public Date now() {
        return new Date();
    }

}
```

Roberto Albertini, Sourcesense

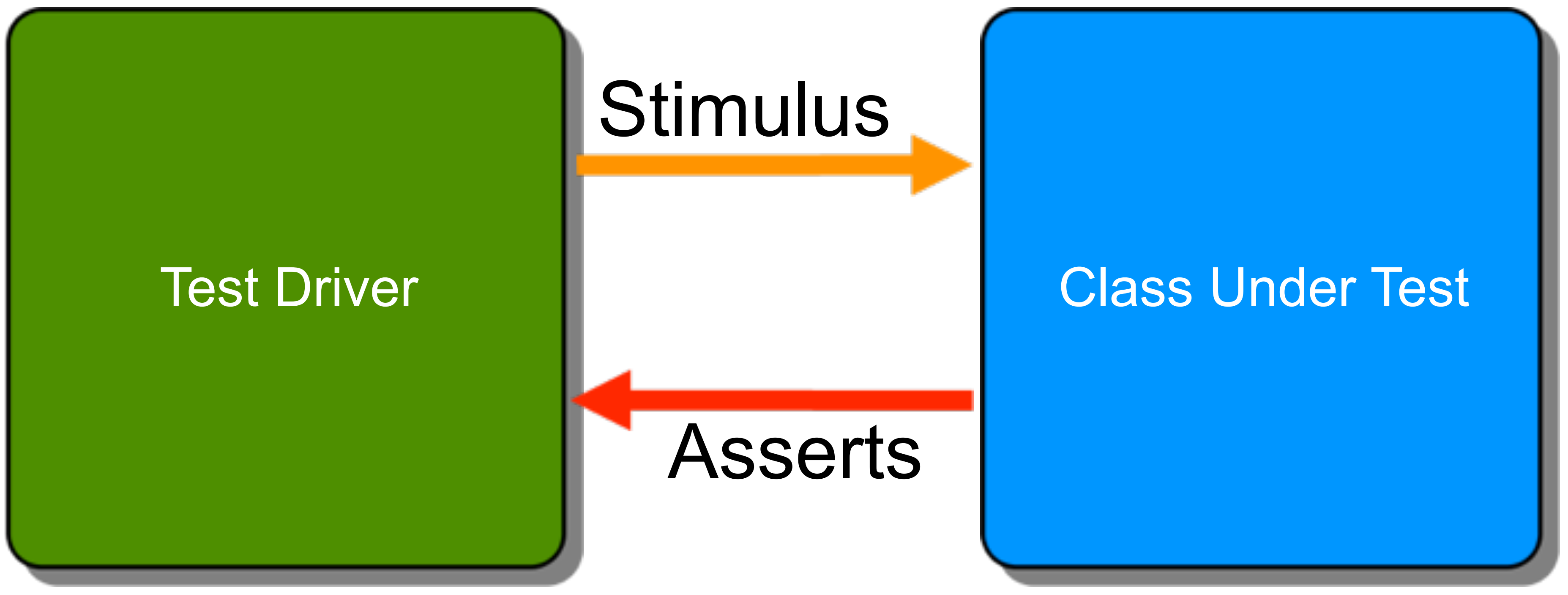
Il codice di test

```java
public class OptionTest {

    @Test
    public void shouldRecognizeExpiredTask() {
        Clock clockAt2009028 =
            new ProgrammableClock(
                dateAt(2009, MAY, 28));

        Date expiration = dateAt(2009, MAY, 27);

        Option option =
                new Option(expiration, clockAt20090528);

        assertTrue(option.isExpired());
    }
}
```

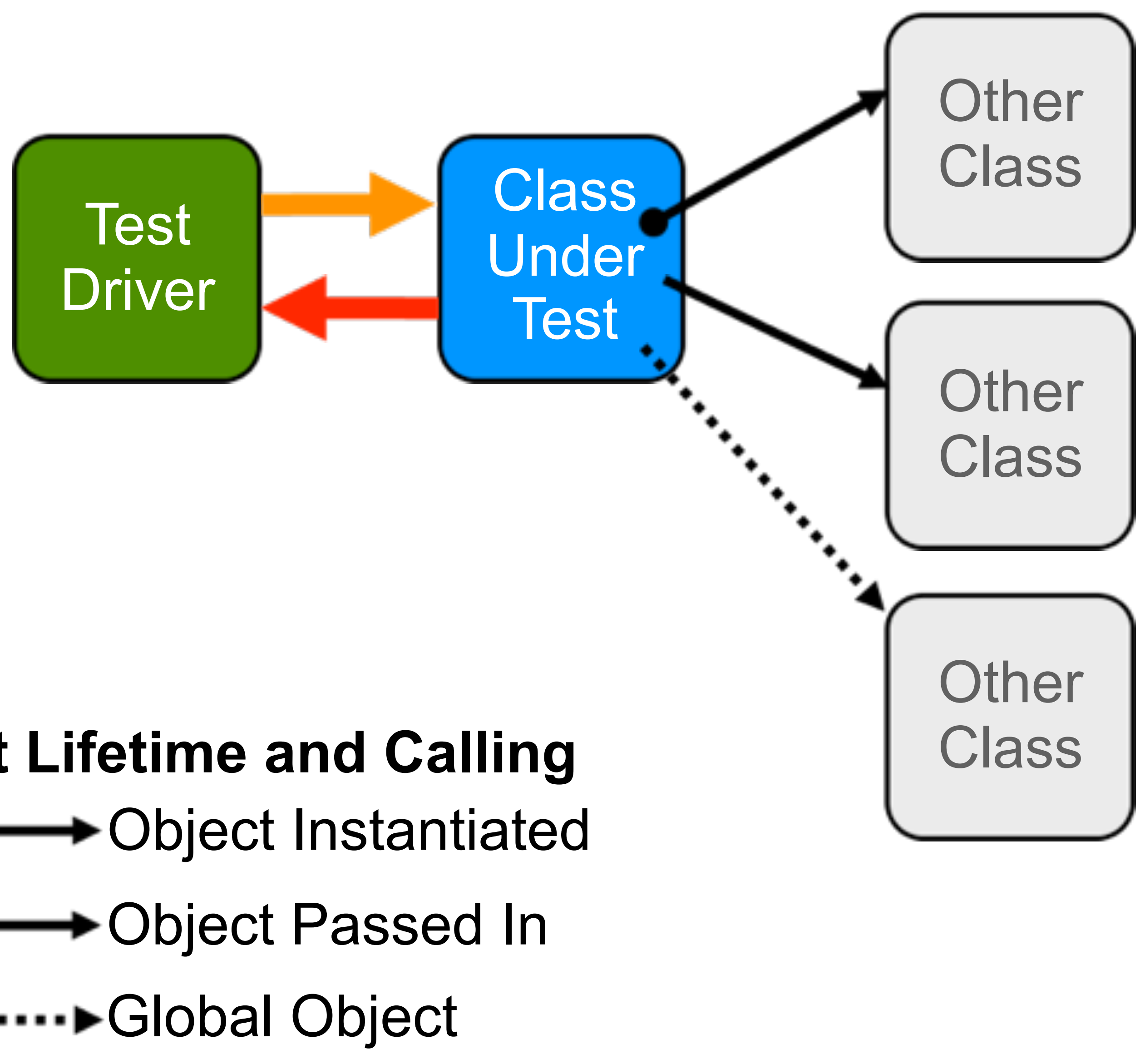
```java
public interface Clock {
    Date now();
}
```

```java
public class ProgrammableClock
                        implements Clock {
    private Date now;

    public ProgrammableClock(Date date) {
        this.now = date;
    }

    public Date now() {
        return now;
    }
}
```

Unit Testing a Class

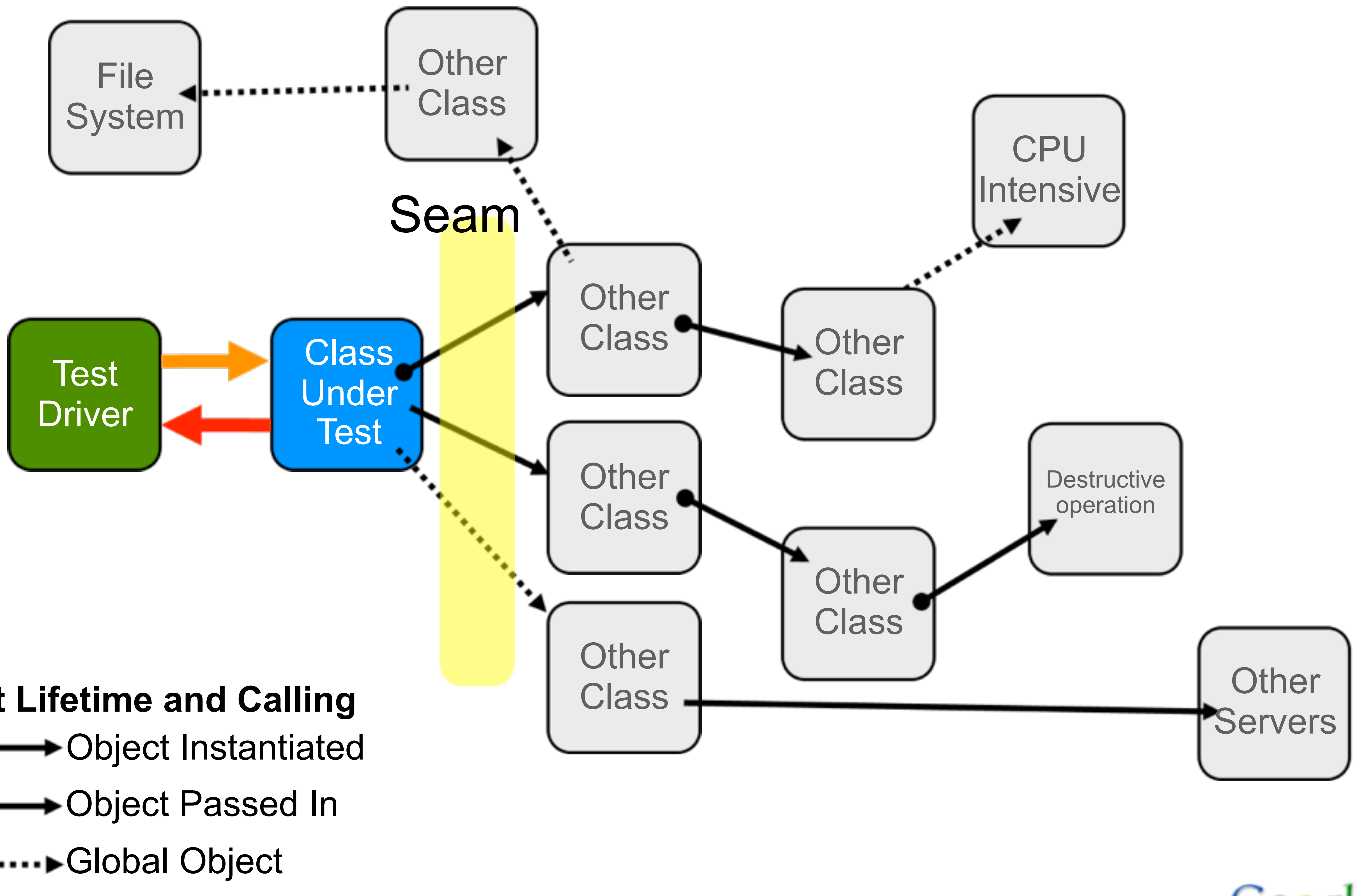


Unit Testing a Class



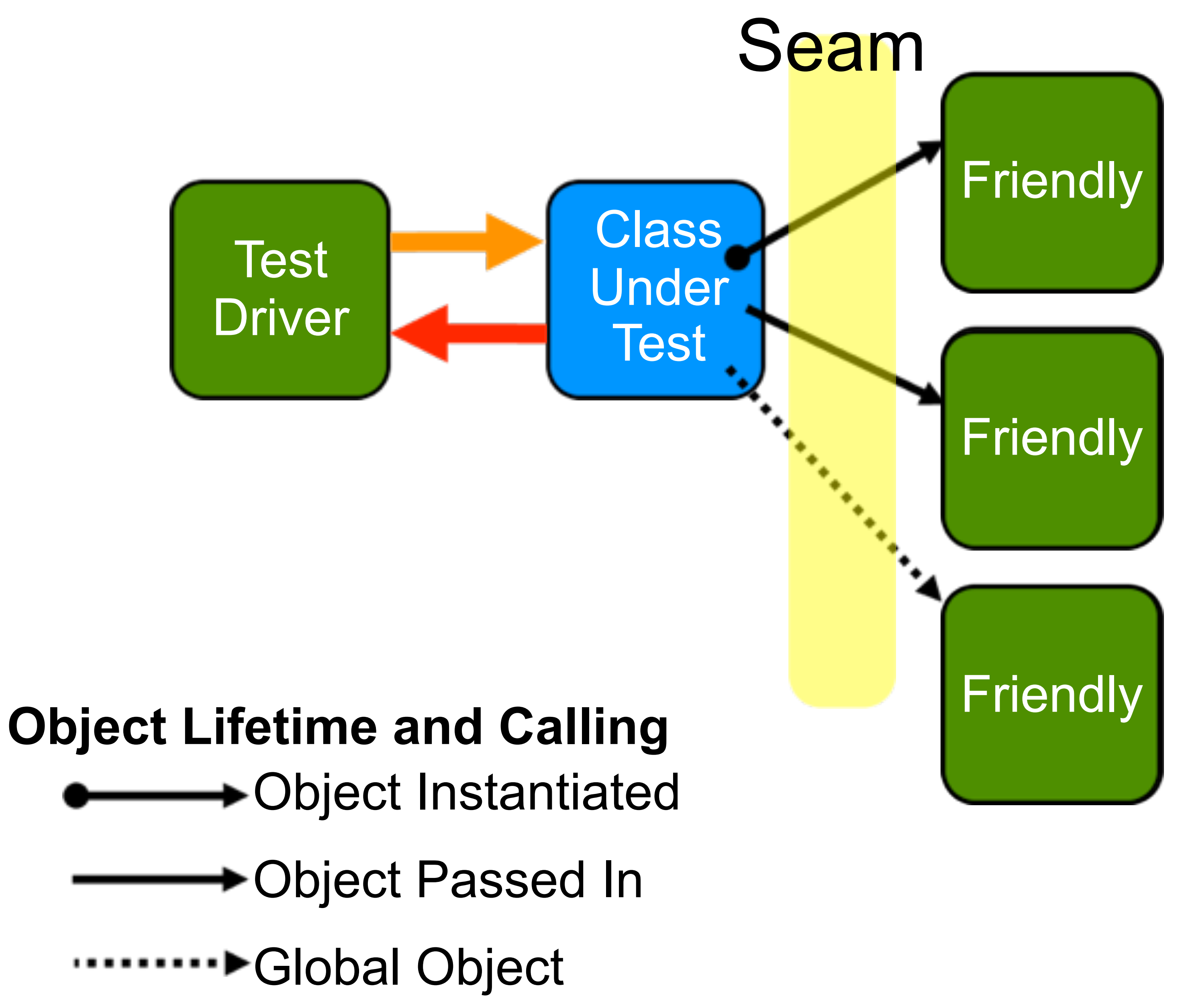
Miško Hevery Google

Unit Testing a Class

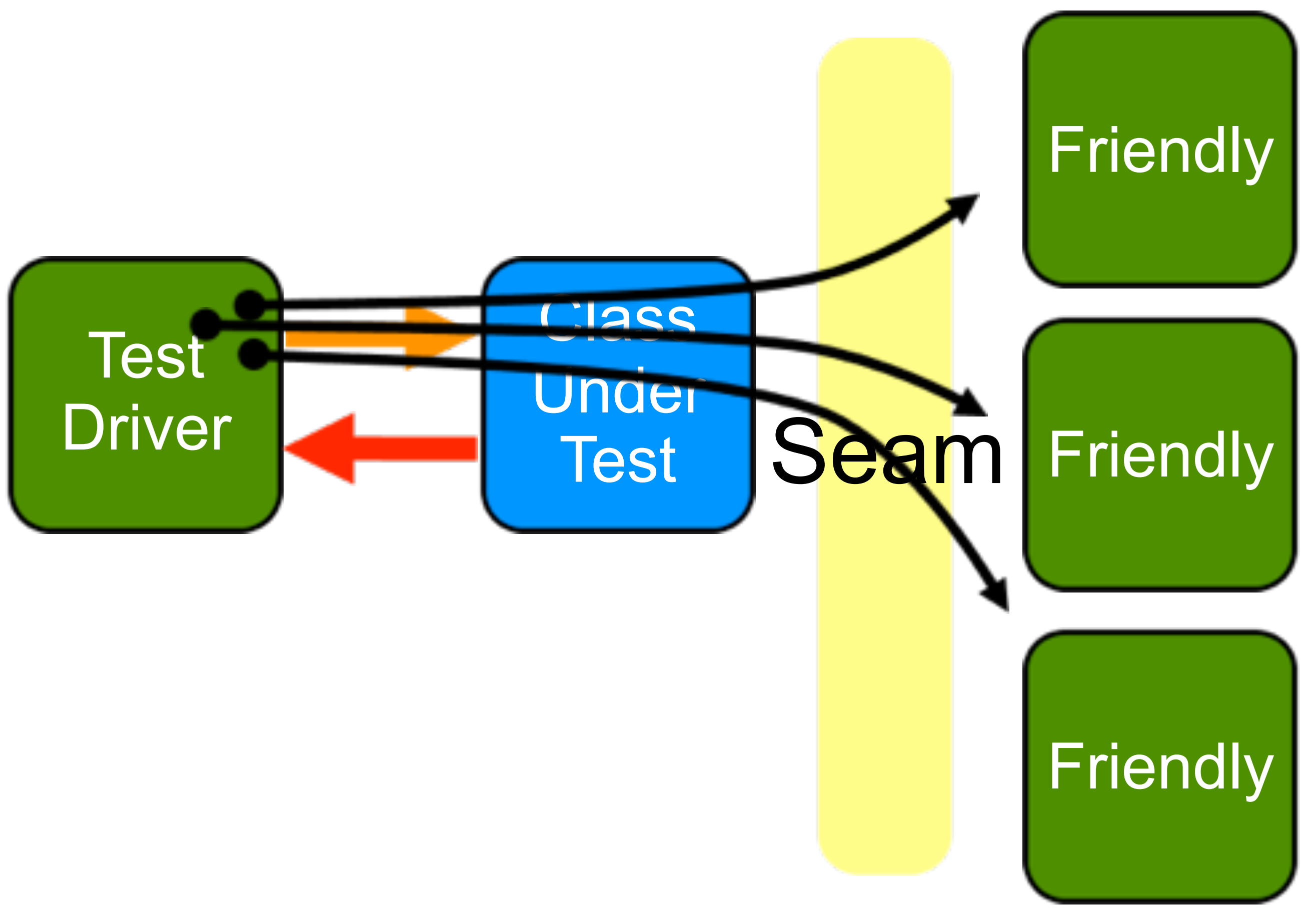


Miško Hevery

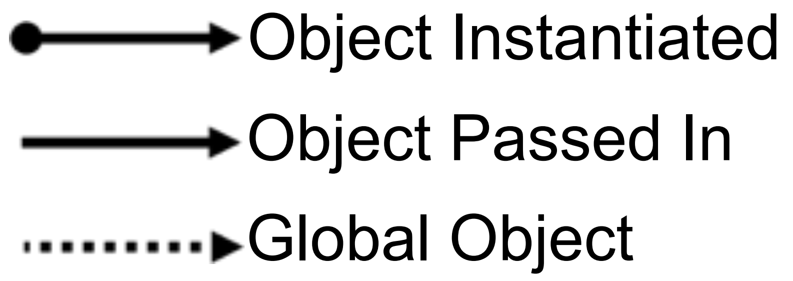
Unit Testing a Class



Unit Testing a Class



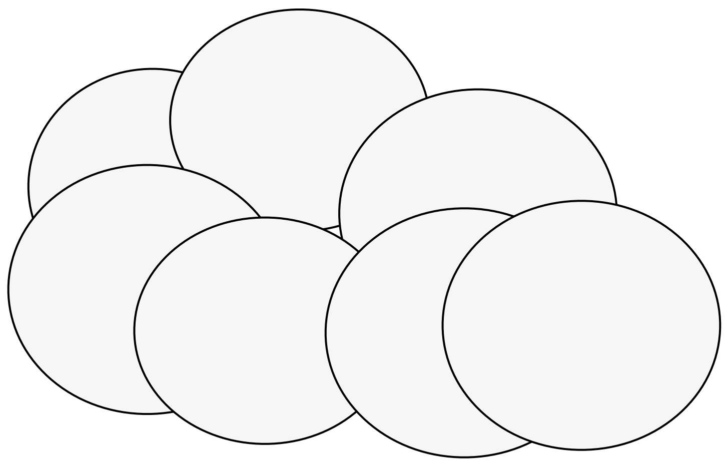
Object Lifetime and Calling



Miško Hevery

Two piles

Pile of Objects
- Business logic
- This is why you're writing code



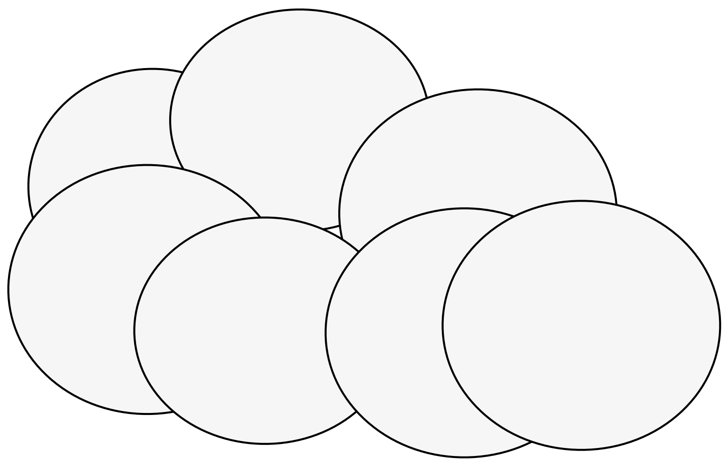
Pile of New Keywords
- Provider<T> objects
- Factories
- Builders
- This is how you get the code you write to work together

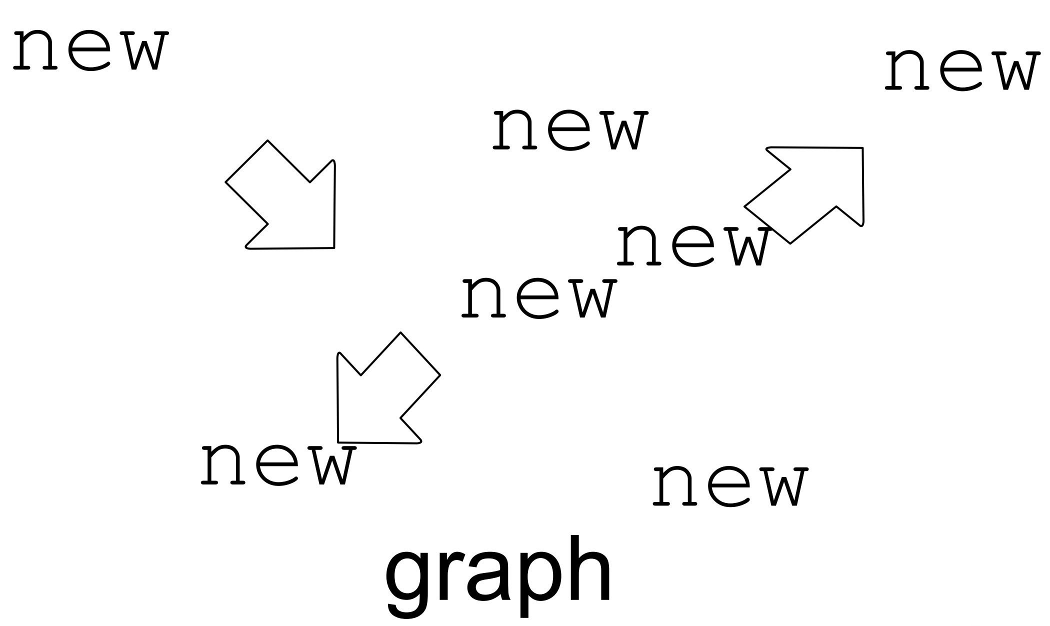
Miško Hevery Google

Two piles

Pile of Objects
- Responsibility is business logic, domain abstractions

Pile of New Keywords
- Responsibility is to build object graphs





La struttura di un *main*

```
public static void main(String[] args) throws Exception {
    // Creation Phase
    Server server = new ServerFactory(args).createServer();

    // Run Phase
    server.start();
}
```

Esercizio: Alla cassa del supermercato

- Compute the total price
- Scan items one at a time
- In any order

Item	Unit	Special
	Price	Price
A	50	3 for 130
B	30	2 for 45
C	20	
D	15	

http://codekata.pragprog.com/

Supermarket checkout: primo test

```
@Test
public void priceForAis50() {
    Checkout checkout = new Checkout( ... );
    assertEquals(0, checkout.total());
    checkout.scan("A");
    assertEquals(50, checkout.total());
    checkout.scan("A");
    assertEquals(100, checkout.total());
}
```

Supermarket checkout: secondo test

```
@Test
public void priceForBIs30() {
    Checkout checkout = new Checkout( ... );
    checkout.scan("B");
    checkout.scan("B");
    assertEquals(60, checkout.total());
}
```

Supermarket checkout: terzo test

```
@Test
public void discountForThreeA() {
    Checkout checkout = new Checkout( ... );
    checkout.scan("A");
    checkout.scan("A");
    checkout.scan("A");
    assertEquals(130, checkout.total());
}
```

Credits

The *Technical Debt* metaphor was coined by *Ward Cunningham*

The *Open-Closed Principle* was coined by *Bertrand Meyer*

Technical debt slides by *Dave Nicolette*
http://www.infoq.com/presentations/TDD-Managers-Nicolette-Scotland

Technical debt example code courtesy of the *Apache Wicket* framework

Anti-IF guy #1 by *Piergiorgio Grossi* - http://pierg.wordpress.com/

Anti-IF guy #2 is *Francesco Cirillo* - http://www.metodiagili.it/

The FizzBuzz example was inspired by a presentation by *Giordano Scalzo*
http://www.slideshare.net/giordano/xpug-coding-dojo-katayahtzee-in-ocp-way

Hard-to-test code slides by *Roberto Albertini* (Sourcesense) and *Miško Hevery* (Google) http://misko.hevery.com/presentations/

The Supermarket Checkout kata is by *Dave Thomas*
http://codekata.pragprog.com